NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

INTERNAL AND EXTERNAL
PERFORMANCE MEASUREMENT METHODOLOGIES
FOR DATABASE SYSTEMS

by

Robert C. Tekampe
Robert J. Watson

June 1984

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Internal and External Performance Measurement Methodologies for Database Systems

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis; June 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Tekampe, Robert C.
Watson, Robert J.

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE
June 1984

13. NUMBER OF PAGES
99

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Performance Measurement; Database System; Database Machine; Benchmarking

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The scope of this thesis is twofold. The first is to provide a methodology for the performance measurement of database systems. The second is the application of this methodology to a specific database system in an attempt to verify the applicability of this methodology and the performance and capacity claims of the database system.

As a methodology, the thesis describes the strategies and locations for the placement of checkpoints, the kinds of

performance data to be collected, the environment for the conduct of the performance measurement and the interpretation of the results. One of the most important contributions of this methodology is its capability to obtain actual measurement overhead making the presentation of truly accurate results possible. As an application of this methodology, we attempt to validate the performance and capacity claims of an experimental multi-backend database system known as MDBS. Surprisingly, these claims have been validated.

Approved for public release; distribution unlimited.

Internal and External Performance Measurement Methodologies for Database Systems

by

Robert C. Tekampe
Captain, United States Marine Corps
B.S.E.E. University of Washington, 1975

and

Robert J. Watson
Captain, United States Marine Corps
B.S.E.E. University of Kansas, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

The scope of this thesis is twofold. The first is to provide a methodology for the performance measurement of database systems. The second is the application of this methodology to a specific database system in an attempt to verify the applicability of this methodology and the performance and capacity claims of the database system.

As a methodology, the thesis describes the strategies and locations for the placement of checkpoints, the kinds of performance data to be collected, the environment for the conduct of the performance measurement and the interpretation of the results. One of the most important contributions of this methodology is its capability to obtain actual measurement overhead making the presentation of truly accurate results possible. As an application of this methodology, we attempt to validate the performance and capacity claims of an experimental multi-backend database system known as MDBS. Surprisingly, these claims have been validated.

TABLE CF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

This work is supported by Contract N00014-84-WR-24058 from the Office of Naval Research to Dr. David K. Hsiac and conducted in the Laboratory for Database Systems Research, located at the Naval Postgraduate School. Dr. Douglas S. Kerr, Adjunct Professor of Computer Science, is the present Director of the Laboratory. The laboratory equipment is supported by DEC, ONR, and NPS.

We would like to thank all those who have supported the MDBS project and who have contributed to the development of this document. In particular, we wish to express our sincere thanks to Drs. Hsiao, Kerr, and Strawser for having provided much of the guidance needed in developing this project. Their time and patience proved invaluable in the development of this performance measurement methodology and its subsequent implementation in the validation of the MDBS claims. A special thanks is to be extended to Steve Demurjian, who as a doctoral candidate in Computer Science at the Ohio State University, is currently doing research at the Naval Postgraduate School. Without his active participation and generous contribution of both knowledge and time, this project would not have been as successful. Lastly, we would like to thank our wives, Peggy Watson and Bobbie Tekampe, for their patience, understanding and support.

I. INTRODUCTION

A. A THESIS OVERVIEW

The scope of this thesis is twofold. The first is to provide a methodology to use in the performance measurement of a database computer. The second is the application of this methodology to a specific database system and the attempt to verify the performance and capacity claims of the target system.

The database system being evaluated is an experimental multi-backend database system known as MDBS. The basic design goal of MDBS is to develop an architecture which spreads the work of the database management among multiple backends. MDBS makes two basic claims in its design. The first is that by increasing the number of backends used as a part of the database computer and by keeping the size of the database constant, the response time of the same transactions is proportionally decreased. The second claim is that by increasing the number of backends and also increasing the size of the database, the response time remains relatively constant.

To conduct the performance measurement of MDBS, various checkpoints and data collections are incorporated into the system. Although all checkpoints and data collections are selected to provide the greatest amount of useful information and to incur the least amount of overhead, some overhead is unavoidable. A quantitative method for measuring the overhead incurred is therefore provided. The performance results of MDBS are then accurately adjusted using the overhead calculation. In this way, a truly accurate measurement of the system may be obtained.

As a methodology, the thesis describes the strategies and locations of the checkpoint placement, the kinds of data on performance collected, the ways in which the performance measurement were conducted and the interpretation of the results. Maybe of greatest importance is the ability to calculate actual measurement overhead allowing for the presentation of truly accurate results.

In this thesis, we will focus our attention on the response time of the work being done by the database system. We will not focus on the throughput. Whereas the throughput is defined as the average number of user requests executed by the system in a second, the response time of a request is the time between the initial issuance of the request by a user and the final receipt of the entire response set of this request by the user [Ref. 1]. Since the majority of the requests processed by a database system are requests for the retrieval of information, another limitation is made to the scope of this thesis. We will focus on the performance measurement of the response time of retrieval requests in MDBS. Hopefully, these evaluations will verify the claims of MDBS and also provide a general methodology for the performance measurement of any database system.

B. THE ORGANIZATION OF THE THESIS

This thesis is organized into six additional chapters beyond this overview. Chapter II describes our performance measurement methodology for database systems. It initially discusses the need for such a methodology and continues with a separate discussion of both the internal and external performance measurements. The chapter then culminates with a discussion of the combination of the two performance measurements, thus providing the methodology to calculate and adjust for internal performance measurement overhead.

Chapter III presents an overview of the target system, MDBS, used to apply the performance measurement methodology. A general discussion is given on the attribute-based data model, the directory tables, the process structure, the message types, and the execution of a retrieve request.

The application of the performance measurement methodology to the target system, MDBS, is presented in Chapter IV. The required modifications to the MDBS software needed to perform the measurements is discussed, along with a discussion of the modifications to the test environment required to control the measurement results. A description of the additional software used for both inter-computer and inter-process message processing measurements is also provided.

Chapter V presents the construction of the test database and the selection of the requests used in the performance measurements. In this chapter, the design of the desired test database is first discussed. Due to system constraints, only a subset of this design is used for testing purposes. The chapter concludes with an analysis of the requests used in the performance measurement.

All the thesis work is brought together in Chapter VI with the presentation of the performance measurement results. Since the goals of this thesis are to verify the performance and capacity claims of MDBS and to provide a methodology for the performance measurement of a database system, only the tests needed to obtain these goals are performed. In the chapter, results are provided for the external and internal performance measurements, and the results of the message processing measurements.

The thesis ends with conclusions in Chapter VII which can be made from the results. It provides a summation for the entire thesis and offers suggestions in future work which needs to be done both with the methodology and with

the measurement of MIES. It is hoped that this thesis will provide a sound methodology for the performance measurements of database systems and also provide a definitive verification of the performance and capacity claims of MDBS.

II. PERFORMANCE MEASUREMENT METHODOLOGY FOR DATABASE SYSTEMS

In this chapter, we present a performance measurement methodology for database systems. The methodology requires the collection of both internal and external performance measurements. The internal performance measurement methodology is the collection of methods and tools which will enable a better understanding of the target system by measuring certain capabilities of that system. In measuring certain capabilities of the system, we focus on the measurement of time spent in individual processes of the target system. The external performance measurement methodology is the collection of methods and tools which will enable the better understanding of the target system by measuring the system as a whole. In measuring the system as a whole, we focus on the measurement of the response time of the target system. The response time in a database system is defined in [Ref. 1] as the time between the initial issuance of the request by a user and the final receipt of the entire response set of this request by the user.

In the rest of this chapter, we begin by examining the need for a database system and the subsequent need to measure the performance of the system. We then discuss a general performance measurement methodology, addressing both internal and external performance measurement as separate issues. Finally, we conclude the chapter with a discussion of the combination of internal and external performance measurement results to provide a complete methodology.

A. THE NEED

The need for a database can best be shown as corresponding to the need for information. A database is a repository for the storage of information on a computer, any item or combination of items of which can be easily accessed in a relatively short timeframe. A businessman may desire all the latest pieces of information to make a management decision. The combat field commander may desire complete, up-to-the-minute reports to arrive at a tactical decision.

But there are performance and capacity problems that must be overcome in providing this information. As an ever increasing amount of information is stored in a database, the response time of the database system increases noticeably. In addition to the increase in the size of the database, there is the effect of increasing the number of users accessing the system and the number of requests to be processed by the system. Thus the user must select between the response time desired and the information desired, a choice the user does not want to and should not have to make. The database system needs to be easily upgraded to accommodate new users and to increase the database size without noticeable change in response time. This is the need for the response-time invariance in a database system.

Another problem is in the timeliness of a response. The database system should offer a dependable, constant return rate for the response to a request. When response time becomes unreasonably long due to the computer workload, the user will be frustrated. A user desires to have every request returned in a timely manner. This is the need for response-time consistency in a database system.

A final problem is to insure that all necessary information is available to the user. Incomplete information is of little use. For example, a user may require all requests to

have a response within a specified timeframe. This requirement often dictates the maximum size of the database and the maximum number of requests. Therefore, an undesireable limitation is placed on the amount of information available due to the limitation on database size. Again, the user is forced into making a tradeoff between the response time and the available information. Nevertheless, despite the response time, such information should be made available to the user on demand. This is the need for availability of information in the database system.

Therefore, not only is there a need for a database system, there is also a need for a database system with the qualities of Invariance, Consistency, and Availability (ICA). But ICA can be present in varying degrees in a database system. The degree of ICA can best be demonstrated by the performance measurement of the database system.

There are two basic types of database systems. The first is an online software database management system that runs on the host computer system. The second is a database machine, which offloads the database functions to a dedicated backend computer. The current trends in database systems involve the design, implementation, and use of database machines [Ref. 1 through 8]. Not only is there an apparent improvement in ICA with a corresponding price per performance advantage, but a database machine can free up resources at the host, provide support for multiple, dissimilar hosts, and increase the security on the database by the physical separation of the database and the host. Due primarily to the trend toward increasing future use of database machines, this thesis will concentrate on the discussion and application of the methodology for measuring the database machines.

A database machine is a database system composed of one or more processors, dedicated to performing the database

management functions. It is indisputable that a database machine is the better of the two types of database systems with regards to providing an increase in security, allowing for multiple host support, and freeing up the host resources. But there still exists the need to demonstrate an improvement in the ICA on a database machine over the ICA provided by a host-resident database system. At the same time, there exists a need to compare the invariance, consistency and availability of several different database machines and software systems. Again, this can best be demonstrated by measureing these systems.

Response-time consistency is more easily achieved in a database machine than in a database system running on the host. Whereas the host must share its resources with a varying workload, the backend can dedicate its resources for database management. Availability frees the Database Administrator from the necessity to make tradeoffs between the size of the database and the response time. The administrator can then load the database with all the necessary information regardless of the database size. To achieve and verify the response time invariance of a database machine, a methodology to measure its effectiveness must be developed.

Thus, the scope of this thesis is to provide a performance measurement methodology for database machines and to verify this methodology by verifying the design claims of a specific database machine, known as MDBS. Again, these claims are related to the quality of response time invariance; that is, to be able to change the size of the database and at the same time maintain constant response time or to hold constant the size of the database with the ability to reduce the response time. Consequently, the measurement of the response time of a database system becomes the focal point of our studies. If the response time can be properly

and accurately measured, the claims of the target system can be verified. Furthermore, the effectiveness of the methodology can also be verified. A proper measurement of the response time can provide a baseline measurement to which other database systems can be compared and thus provide a price-performance comparison of various systems. This thesis provides an overhead-free performance-measurement methodology and applies this methodology to verify the claims of an experimental database machine.

B. THE APPROACH

In this section, we discuss a general methodology to be used in the performance measurement of a database system. This methodology is general and can be applied to any other database system. We first discuss the internal performance measurement. This includes the design considerations, the software engineering criteria and the application of the methodology to a particular system. Then we present a discussion of the external performance measurement, again discussing the design considerations, the software engineering criteria, and the application of the methodology to a particular system.

1. A Methodology for Internal Performance Measurement

The goal of the internal performance measurement methodology is to provide methods and tools which will enable us to better understand the target system by measuring certain aspects of that system. A complete understanding of how the system performs internally may lead to design modifications or to fine-tuning of the system for better performance. The internal performance measurement tools should be unobtrusive to the user, available when necessary, yet out of the way when not required. They should

be integrated with the target system to produce a smooth transition between target system operation and the operation of the tool. In the first part of this section, we address the design considerations of internal performance measurement methods. Next, we discuss certain software engineering criteria which are applicable to the design of good measurement tools. Finally, we explore the application of the internal performance measurement methodology to a particular system.

a. Design Considerations

Internal performance measurement relies on checkpoints internal to the database system software. A checkpoint is defined as a procedural invocation inserted into the system's flow of control to call the performance measurement routines which are used for the data collection. System overhead is introduced as each checkpoint is added to the target system. Additionally, measurement software is required to process the checkpoint data in a manner compatible with the existing target system software. That is, a certain portion of the measurement software must be integrated with the target system software to handle events such as data storage, message passing, and information processing that relate to the checkpoint data. Finally, the existing target system software may require additional lines of code to handle new cases introduced by the measurement system.

In most external performance measurement, overhead is negligible. However, internal measurement routines add significant overhead to the database system which cannot be disregarded. For internal measurement, we must discover ways to reduce the overhead generated by the measurement software. We must also be able to measure the overhead which cannot be eliminated, so that the measurements can be adjusted accordingly. A very important requirement is that

the existing target system must maintain the capability of running unimpeded by the additional measurement software.

Consideration must be given to the level in the target system where checkpoints may be placed. Some possible levels are at the very high level, i.e., the system level, the high level, i.e., the program level, the medium level, i.e., the subroutine level, and the low level, i.e., the subroutine segment level. Whereas external performance measurement only places checkpoints at the very high level, internal performance measurement places checkpoints below that level. Checkpoints must be placed at a level which produces data in sufficient detail to provide the user with a basic understanding of the system's performance characteristics. Checkpoints should not be placed at a level so low as to overwhelm the user with detailed data or to interfere significantly with system performance.

For internal performance measurement, the user should have the capability to access selected data out of a range of possible choices. The user should not be required to receive information about processes which are not of current interest. The interface should be easy to use and should not distract the user from his primary goal of understanding the database system by requiring the user to remember the unique syntax or semantics of the test interface. The collected measurements should be made accessible to automated processing routines for data reduction.

b. Software Engineering Criteria

Measurement software should be designed using modern software engineering methods. The resulting software should be understandable, maintainable, reliable and compatible with the target system. Certain software engineering methods are of particular interest. These methods are modularization, user-friendliness, data abstraction, and simplicity.

For modularity, the measurement programs should be hierarchically structured with well-defined interfaces. The measurement modules should be reusable throughout the target system. Modularity allows the system to be easily extended to checkpoints not considered in the initial specifications. The test interface should present an easy-to-use method for obtaining test data. It should automatically aggregate data while still allowing the user to access raw data. The user should not have to remember the specific syntax and semantics of the test interface. Data abstraction should be used so that subsequent program modifications do not result in extensive reprogramming. An appropriate choice of primitives (data structure and operations) will allow for easy change and produce less system overhead. The measurement system should be user-friendly. In addition to obeying the simplicity principle, the test interface should be forgiving, i.e., system should not crash on bad input, provide readable error diagnostics, anticipate errors, and guard against those errors.

c. Issues in the Application to Database Systems

Application of the internal performance measurement methodology to a particular database system requires that the evaluator understand certain aspects of the target system. The evaluator must understand the programming language used to construct the database system, and the structure and operation of the database system. The evaluator must be prepared to overcome obstacles presented by the target system in the course of the implementation of the performance measurement.

A thorough understanding of the programming language is necessary to successfully integrate checkpoints and data collection programs into the existing software structure. One must be familiar with the data structures,

control structures, naming conventions, and parameter-passing mechanisms of the language, in order to implement the measurement programs efficiently and to minimize their overhead. Knowledge of the language syntax reduces programming errors and speeds implementation of the measurement tools.

For effective internal performance measurement, checkpoints must be correctly placed in the database system. Incorrectly placed checkpoints increase overhead and degrade performance measurement by providing useless data to the user. The evaluator must possess sufficient knowledge of the target system to allow for the correct placement of checkpoints. This provides the smooth integration of data collection programs, data processing programs and data transfer programs into the existing database system.

Chances are that the target system, when initially designed, was not designed with internal performance measurement in mind. Instead, the target system was designed to process all requests efficiently. Integration of the internal performance measurement routines may affect the target system in unexpected ways. Let us consider two examples of such ways. First, in a message-passing system, messages generated by the measurement programs may require modifications to the existing database system so that test messages will not be confused with the messages of the database system. Second, the volume of information generated by the measurement programs may overload selected sections of the target system. The evaluator of the performance measurement routines must be prepared for such contingencies. By using the knowledge of the programming language along with the knowledge of the database system, the evaluator must be prepared to offer solutions to the database administrator on how to gracefully integrate the performance measurement mechanisms into the target system with proper modification and without overload.

2. A Methodology for External Performance Measurement

The goal of external performance measurement is to provide a collection of methods and tools which will enable us to better understand the target system by measuring the system as a whole. In this way we can measure the total work being done by the database system. We focus on measuring the response time of the system, the elapsed time between the issuance of a request and the receipt of the response to the request.

Internal performance measurement has been shown to be beneficial in the fine-tuning of a system, and in the microscopic examination of the work being performed by the system. External measurement provides a quantitative measurement of the system from a macroscopic view. This allows for the comparison of database systems. In the first part of the section, we discuss the design considerations of the external performance measurement methods. Next, we present the software engineering criteria for external performance measurement. Lastly, we show the application of the external performance measurement to a system.

a. Design Considerations

External performance measurement should have negligible overhead, i.e., the response time with external performance measurement should be the same as the response time without measurement being performed. This is in fact the case. The reason that the overhead is negligible is that only two timing checkpoints need to be made. These timing checkpoints are placed at the beginning of a request and the end of the response to the request, thus providing the elapsed time of the response for a request. The timing checkpoints need the system time at the start and completion of the request. The checkpoints are placed at the very high

level to insure a complete measurement of the total elapsed time.

There are other issues that must be considered to insure that the system being evaluated is as 'pure' as possible. First, the system should retain only those code and messages required for the running of the system. Messages and code incorporated into the system for the design or debugging of the system should be removed. Second, the system should not contain unnecessary software tools designed to aid the measurement, such as those used to create a test database. Such tools should remain in software exterior to the actual database system.

An obvious consideration is to insure that no human interaction is involved in the timings. The system software, not the reaction time of the user, is being timed. Therefore, the timer should start immediately after a user releases the request. The timer should stop immediately prior to the display on the selected output device. The reason for stopping the timer prior to display is due to the varying delays caused by the output devices. The speed of an output device should not be included into the system timing results.

The final issue involving the placement of external performance measurement checkpoints is whether to embed the timer code in the system or to call a timer routine outside the system. A call to a timer routine incurs unwanted timing delays, adding to the impurity of a system. If the timer code is embedded, it can be made to appear that the system code being tested is embedded in the timer code, i.e., placing the timer initialization code just prior to the point of the request by the user and the timer finalization code just subsequent to the display on the output device. With these considerations, an optimal placement of checkpoints can be selected to take external performance timings.

b. Software Engineering Criteria

Unlike internal performance-measurement software which uses software design methodologies, the external performance-measurement software uses software design tools. In [Ref. 9], a full description is provided of the necessary external performance-measurement tools. These tools include a test-file generation package, a database load subsystem, and a request generation package.

The purpose of the test-file generation package is to create a test database. This allows for the easy creation of a database containing the desired parameters to be evaluated. The database load subsystem must properly load the files created in the generation package. This includes the creation of directories for the test database. The request generation package is used to create and execute test requests, and provides for easy variance in the types and complexity of requests. This package also archives the requests for later use. Using these tools, the external performance timings of the database system under measurement can be easily obtained.

c. Issues in the Application of the Methodology

The ease with which external performance measurement can be performed on a database system can vary. There are two important considerations: the language in which the system is written and the degree of software engineering used in the database system design.

The language needs to be readable and to compliment proper documentation of the system. This will facilitate an understanding of the system by the system evaluator. The language must also be powerful enough to easily incorporate system commands, such as requests for the system time. A language, such as C, has these capabilities, being

primarily designed for system programming. C is a high-level language, that is both powerful and portable. Although the support software tools such as database load can be implemented in a language other than the language in which the database system was written, the evaluator needs to be familiar with several different languages if several different database systems are to be evaluated.

The degree of software engineering used in the database system design will most definitely facilitate any external performance measurement to be done. If the database system was hierarchically designed using modularity, knowledge of the internal workings of the system by the evaluator will be minimal. Only the upper level in the hierarchy need to be studied for the proper placement of the checkpoints. External measurement only requires a macro knowledge of the system. This is to insure that the checkpoints are indeed properly placed at the very high level.

C. THE COMBINATION OF INTERNAL AND EXTERNAL PERFORMANCE MEASUREMENTS

Separately, internal and external performance measurements provide a wealth of information to the evaluator. Internal performance measurement provides the timings and data collections of individual processes in the database system. External performance measurement provides the elapsed time for the complete request. Yet, when the two methodologies are combined, there is a synergistic effect to the amount of information available to the evaluator.

The combination of internal and external performance measurements is natural. There are benefits to be gained for one from the other. For example, we can determine the overhead incurred when using internal performance measurement; first, using the external checkpoint, we collect the

elapsed time for processing a particular request. This time is then compared to the elapsed time of the request when both internal and external checkpoints are enabled. The difference in the elapsed times of these two measurements provides an exact measurement of the overhead incurred by the internal performance measurement software for this request.

On the other hand, we can use the internal performance measurement timings to interpret the external performance measurement timings. In particular, if a request takes many hundredths of a second as a result of external performance measurement, the evaluator would want to determine the precise distribution of the work. Internal performance measurement can answer these questions. By combining the two measurements, the whole of the measurement results is more meaningful and useful than the individual results.

In the following chapter, the target system, i.e., MDBS is described. This is the system selected to be evaluated using the internal and external performance measurement methodologies presented in this chapter.

III. THE MULTI-BACKEND DATABASE SYSTEM (MDBS)

In this chapter we discuss the configuration and theory of operation of the multi-backend database system (MDBS). This chapter has been extracted from papers and reports which have been written on MDBS [Ref. 6, 10, 11, 12].

MDBS uses two or more identical minicomputers and their disk systems to provide a centralized database system with support for multiple, dissimilar hosts. One minicomputer functions as the controller. User access is accomplished through a host computer which in turn communicates with the controller. Multiple minicomputers and their disks are configured in parallel to serve as backends. The original design and analysis of MDBS is due to J. Menon [Ref. 1, 2]. The implementation and new design efforts are documented in [Ref. 3 through 6]. The database is distributed across all of the backends. The database management functions are replicated in each backend.

As shown in Figure 4.1, the controller and the backends are connected by a broadcast bus. When a transaction is received from the host computer, the controller broadcasts the transaction to all the backends. Each backend has a number of dedicated disk drives. Since the data is distributed across the backends, a transaction can be executed by all backends concurrently. Each backend maintains a queue of transactions and schedules requests for execution independent of the other backends, in order to maximize its access operations and to minimize its idle time. The controller does very little work. It is responsible for broadcasting, routing, and assisting in the insertion of new data. The backends do most of the database operations. Presently, MDBS is fully operational with a VAX 11/780 as

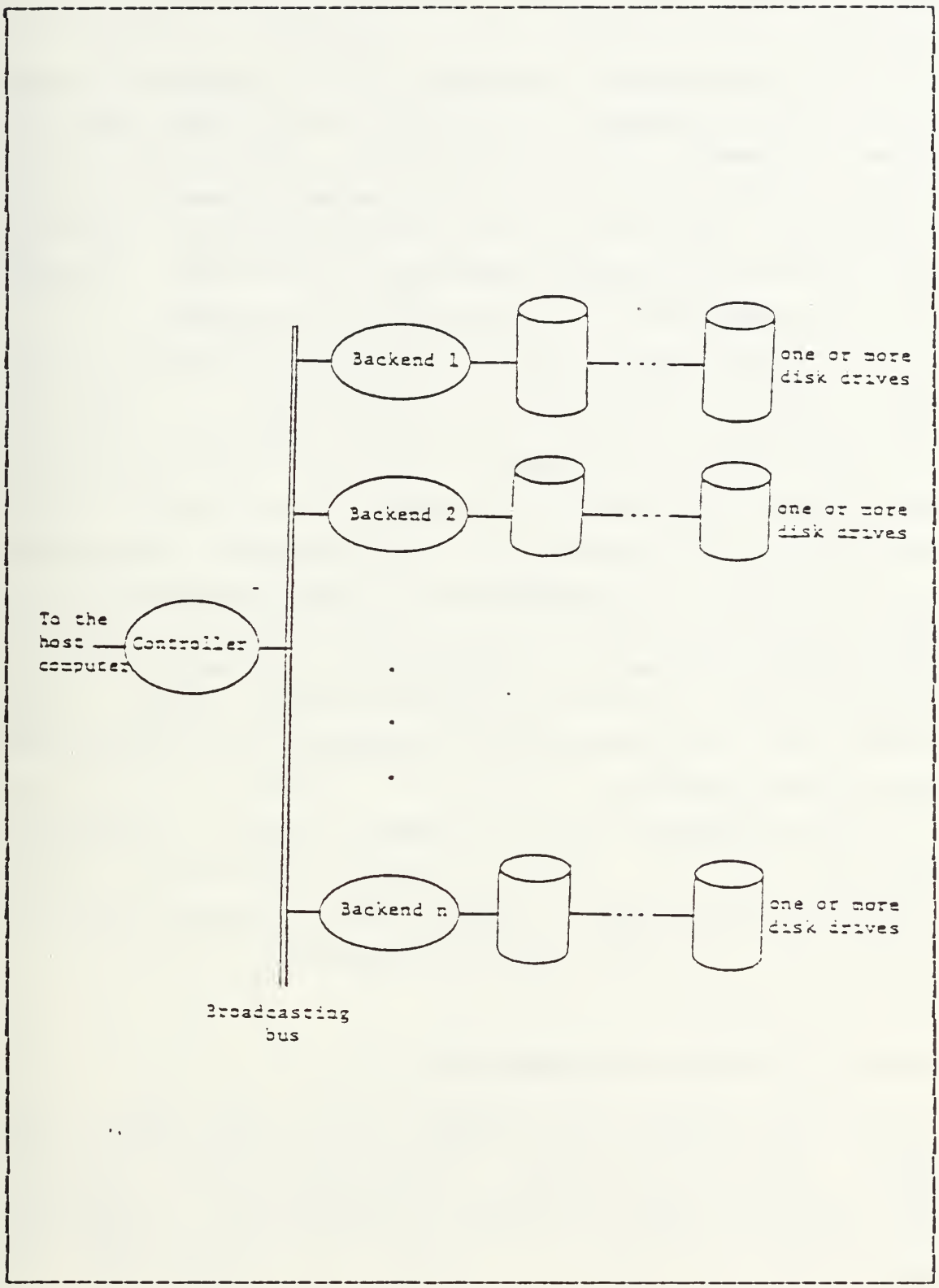

Figure 4.1 The MDBS Structure.

the controller and two PDP 11/44s and their disks as the backends.

MDBS is a message-oriented system. In a message-oriented system, each process corresponds to one system function. These processes, then, communicate among themselves by passing messages. User requests are passed between processes as messages. The message paths between processes are fixed for the system. The MDBS processes are created at system start time and exist until the system is stopped.

MDBS is designed to perform the primary database operations, INSERT, DELETE, UPDATE, and RETRIEVE. Of these four database operations, only the retrieval operation will be of concern to us in this thesis. The syntax and semantics of the retrieve operation is discussed in Chapter V. Users access MDBS through the host by issuing either a request or a transaction. A transaction is a set of requests. A request is a primary operation along with a qualification. A qualification is used to specify the information of the database that is to be accessed by the request. More complete definitions of the MDBS terminology can be found in the following section.

In the remainder of this chapter we first discuss the directory structure. Next, we provide an overview of the process structure. Then, a presentation of the message types is provided. Lastly, we trace the execution sequence of a retrieve request.

A. THE ATTRIBUTE-BASED DATA MODEL

In this section we discuss the attribute-based data model. Next we provide some definitions in order to discuss MDBS directory data. We conclude this section by describing the tables necessary to maintain the MDBS directory information.

In the attribute-based data model, data is modeled with the constructs: database, file, record, attribute-value pair, directory keyword, directory, record body, keyword predicate, and query. Informally, a database consists of a collection of files. Each file contains groups of records which are characterized by a unique set of directory keywords. A record is composed of two parts. The first part is a collection of attribute-value pairs or keywords. An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. As an example, <POPULATION, 25000> is an attribute-value pair having 25000 as the value for the population attribute. A record contains at most one attribute-value pair for each attribute defined in the database. Certain attribute-value pairs of a record (or a file) are called the directory keywords of the record (file), because either the attribute-value pairs or their attribute-value ranges are kept in the directory for addressing the record (file). Those attribute-value pairs which are not kept in the directory for addressing the record (file) are called non-directory keywords. The rest of the record is textual information, which is referred to as the record body. An example of a record is shown below.

(<FILE, Census>, <CITY, Monterey>, <POPULATION, 25000>,
{ Temperate climate })

The angle brackets, <,>, enclose an attribute-value pair, i.e., keyword. The curly brackets, {,}, include the record body. The first attribute-value pair of all records of a file is the same. In particular, the attribute is FILE and the value is the file name. A record is enclosed in the parenthesis. For example, the above sample record is from the Census file.

The database is accessed by indexing on directory keywords using keyword predicates. A keyword predicate is a three-tuple consisting of an attribute, a relational operator (=, ≠, ≥, ≤, >, <), and an attribute value, i.e., POPULATION ≥ 20000 is a keyword predicate. More specifically, it is a greater-than-or-equal-to predicate. Combining keyword predicates in disjunctive normal form characterizes a query of the database. The query

(FILE = Census and CITY = Monterey) or
(FILE = Census and CITY = San Jose)

will be satisfied by all records of the Census file with the CITY of either Monterey or San Jose. For clarity, we also employ parentheses for bracketing predicates in a query.

Recall that in MDBS there are four types of requests which correspond to the four primary database operations. An example of a retrieve request would be:

RETRIEVE (FILE = Census and POPULATION > 10000) (CITY)

which retrieves the names of all those cities in the Census file whose population is greater than 10000. Notice that the qualification component of a retrieve request consists of two parts, the query of two predicates (FILE = Census and POPULATION > 10000) and the target list (CITY). The query specifies which records of the database are to be retrieved. The target list specifies the attribute-value(s) to be returned to the user. A user may wish to treat two or more requests as a transaction. In this situation, MDBS executes the requests of a transaction without permuting them, i.e., if T is a transaction containing the requests <R1><R2>, then MDBS executes the request R1 before request R2. Finally, we define the term traffic-unit to represent either a single request or a transaction in execution.

B. THE DIRECTORY TABLES

To manage the database (often refered to as user data), MDBS uses directory data. Directory data in MDBS corresponds to attributes, descriptors, and clusters. An attribute is used to represent a category of the user data; e.g., POPULATION is an attribute that corresponds to actual populations stored in the database. A descriptor is used to describe a range of values that an attribute can have; e.g, (10001 ≤ POPULATION ≤ 15000) is a possible descriptor for the attribute POPULATION. The descriptors that are defined for an attribute, e.g., population ranges, are mutually exclusive. Now the notion of a cluster can be defined. A cluster is a group of records such that every record in the cluster satisfies the same set of descriptors. For example, all records with POPULATION between 10001 and 15000 may form one cluster whose descriptor is the one given above. In this case, the cluster satisfies the set of a single descriptor. In reality, a cluster tends to satisfy a set of multiple descriptors.

Directory information is stored in three tables: the Attribute Table (AT), the Descriptor-to-Descriptor-Id Table (DDIT) and the Cluster-Definition Table (CDT). The Attribute Table maps directory attributes to the descriptors defined

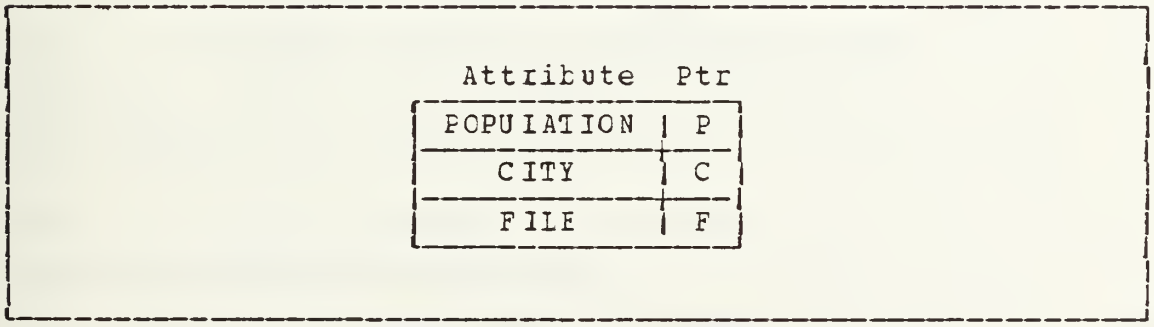

Figure 4.2 An Attribute Table (AT).

on them. A sample AT is depicted in Figure 4.2. The Descriptor-to-Descriptor-Id Table maps each descriptor to a unique descriptor id. A sample DDIT is given in Figure 4.3. Note that the pointer shown in Figure 4.3 is not actually in the DDIT table but is shown here for clarity to relate back

	Descriptor	Id
P->	0 ≤ POPULATION ≤ 50000	D11
	50001 ≤ POPULATION ≤ 100000	D12
	100001 ≤ POPULATION ≤ 250000	D13
	250001 ≤ POPULATION ≤ 500000	D14
C->	CITY = Cumberland	D21
	CITY = Columbus	D22
F->	FILE = Employee	D31
	FILE = Census	D32

Dij: Descriptor j for attribute i.

Figure 4.3 A Descriptor-to-Descriptor-Id Table (DDIT).

to the AT table of Figure 4.2. The Cluster-Definition Table maps descriptor-id sets to cluster ids. Each entry consists of the unique cluster id, the set of descriptor ids whose descriptors define the cluster, and the addresses of the records in the clusters. A sample CDT is shown in Figure 4.4. Thus, to access the directory data, we must access the AT, DDIT, and CDT.

One of the key concepts used when designing the test database (see Chapter V.) is defining the descriptors which are specified in the directory attributes. Thus, we provide a brief introduction to the three classifications of descriptors. A type-A descriptor is a conjunction of a

Id	Desc-Id Set	Addr
C1	D11,D21,D31	A1,A2
C2	D14,D22,D32	A3

Figure 4.4 A Cluster-Definition Table (CDT).

less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. An example of a type-A descriptor is as follows:

((POPULATION ≥ 10000) and (POPULATION ≤ 15000)).

A type-B descriptor consists of only an equality predicate. An example of a type-B descriptor is:

(FILE = Census).

Finally, a type-C descriptor consists of the name of an attribute. The type-C attribute defines a set of type-C sub-descriptors. Type-C sub-descriptors are equality predicates defined over all unique attribute values which exist in the database. For example, the type-C attribute CITY forms the type-C sub-descriptors

(CITY=Cumberland), (CITY=Columbus)

where "Cumberland" and "Columbus" are the only unique database values for the CITY.

C. THE PROCESS STRUCTURE

Currently, MDBS does not communicate with a host machine. The absence of this communication requires that the

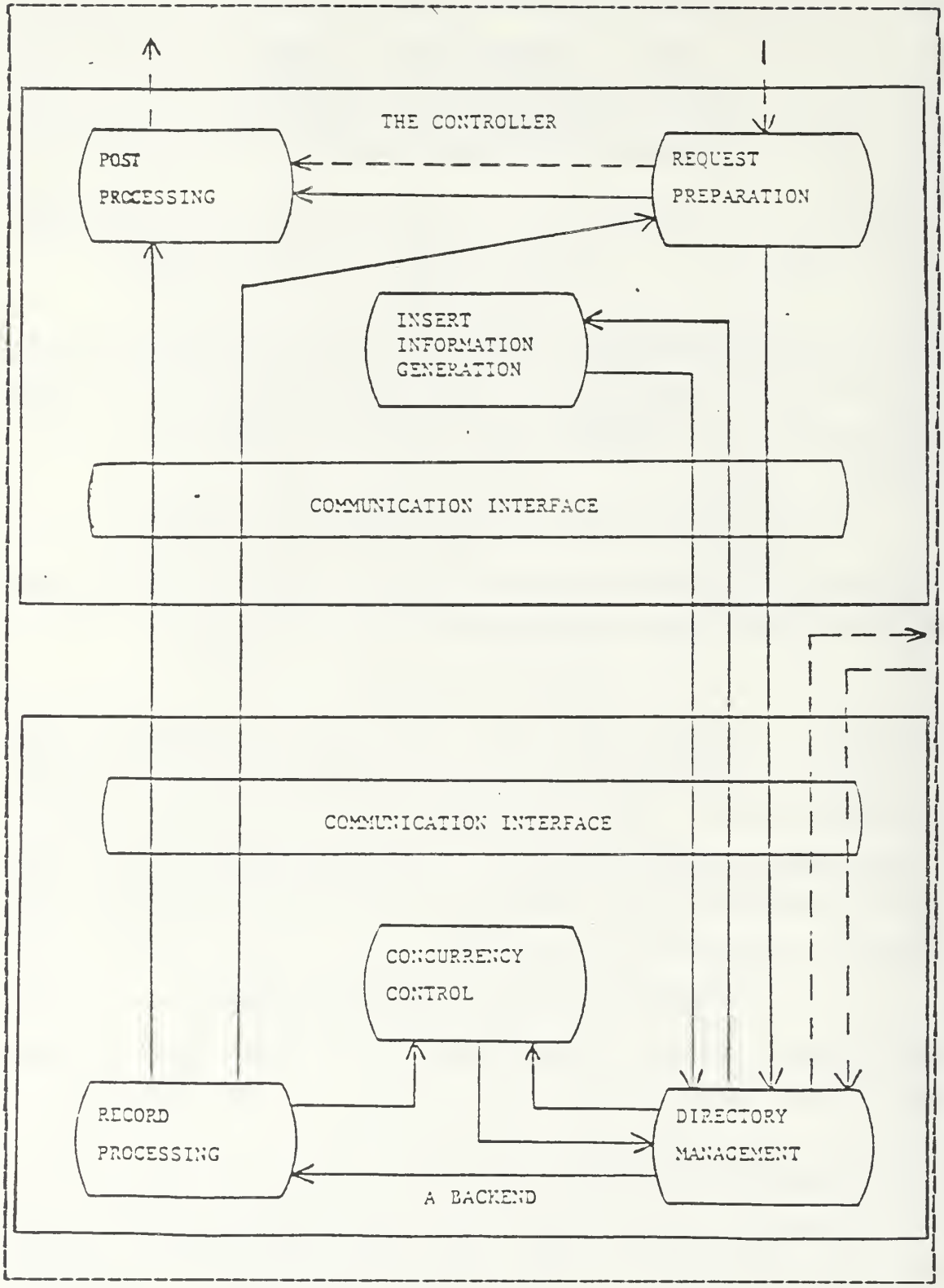

Figure 4.5 The MDBS Process Structure.

test interface process, the process used to interact with MDBS, be placed in the MDBS Controller. The current implementation of MDBS does not utilize a broadcast bus. Instead, MDBS utilizes parallel communications links (PCIs) to emulate a broadcast bus. Both the controller and the backends contain processes to interface with the PCLs for intercomputer message passing. These processes, while necessary to interface with the PCLs, are not part of MDBS and will not be discussed further. Figure 4.5 provides an overview of the MDBS Process Structure.

1. The Processes of the Controller

The controller is composed of three processes: Request Preparation, Insert Information Generation, and Post Processing. Request Preparation receives, parses and formats a request (transaction) before sending the formatted request (transaction) to the Directory Management process in each backend. Insert Information Generation is used to provide additional information to the backends when an insert request is received. Since the data is distributed, the insert only occurs at one of the backends. Thus it must determine the backend at which the insert will occur, along with the cluster and descriptor ids for the insert. Post Processing is used to collect all the results of a request (transaction) and forward the information back to the host computer.

2. The Processes of Each Backend

Each backend is also composed of three processes. They are of course different from the controller processes. They are: Directory Management, Concurrency Control, and Record Processing. Directory Management performs Descriptor Search, Cluster Search, and Address Generation. Descriptor Search determines the descriptor ids that are needed for a

request. Cluster Search finds the cluster ids. Address Generation determines the secondary storage addresses necessary to access the clustered records. Concurrency Control determines when the request can be executed. Record Processing performs the operation specified by the request.

D. THE MDBS MESSAGE TYPES

In this section we describe the MDBS message-passing facilities first described in [Ref. 13]. In the MDBS message-passing facilities there are 31 message types and one general message format (shown in Figure 4.6). This same

Message Type	(a numeric code).
Message Sender	(a numeric code).
Message Receiver	(a numeric code).
Message Text	(an alphanumeric field terminated by an end of message marker).

Figure 4.6 The General Message Format.

format is used for each of the three message passing facilities, namely, messages within the controller, messages within a backend, and messages between computers. Messages between computers are divided into two classes, messages between backends, and messages between the controller and the backends. Figure 4.7 describes each of the MDBS message types. Figure 4.8 describes the abbreviations used.

	MESSAGE-TYPE NUMBER AND NAME	SRC	DEST	PATH
1	TRAFFIC UNIT	1 HOST	1 REQP	1 HC
	REQUEST RESULTS	PP	HOST	CH
	NUMBER OF REQUESTS IN A TRANSACTION	REQP	PP	C
	AGGREGATE OPERATORS	REQP	PP	C
5	REQUESTS WITH ERRORS	5 REQP	5 PP	5 C
	PARSED TRAFFIC UNIT	REQP	DM	CE
	NEW DESCRIPTOR ID	IIG	DM	CE
	BACKEND NUMBER	IIG	DM	CE
	CLUSTER ID	DM	IIG	BC
10	REQUEST FOR NEW DESCRIPTOR ID	10 DM	10 IIG	10 BC
	BACKEND RESULTS FOR A REQUEST	RECP	PP	BC
	BACKEND AGGREGATE OPERATOR RESULTS	RECP	PP	BC
	RECORD THAT HAS CHANGED CLUSTER	RECP	REQP	BC
	RESULTS OF A RETRIEVE OR FETCH CAUSED BY AN UPDATE	RECP	REQP	BC
15	DESCRIPTOR IDS	15 DM	15 DMs	15 BB
	REQUEST AND DISK ADDRESSES	DM	RECP	B
	CHANGED CLUSTER RESPONSE	DM	RECP	B
	FETCH	DM	RECP	B
	CLI AND NEW VALUES OF ATTRIBUTE BEING MODIFIED	RECP	DM	B
20	TYPE-C ATTRIBUTES FOR A TRAFFIC UNIT	20 DM	20 CC	20 B
	DESC-ID GROUPS FOR A TRAFFIC UNIT	DM	CC	B
	CLUSTER IDS FOR A TRAFFIC UNIT	DM	CC	B
	RELEASE ATTRIBUTE	DM	CC	B
	RELEASE ALL ATTRIBUTES FOR AN INSERT	DM	CC	B
25	RELEASE DESCRIPTOR-ID GROUPS	25 DM	25 CC	25 B
	ATTRIBUTE LOCKED	CC	DM	B
	DESCRIPTOR-ID GROUPS LOCKED	CC	DM	B
	CLUSTER IDS LOCKED	CC	DM	B
29	NO MORE GENERATED INSERTS	RECP	REQP	BC
29	NO MORE GENERATED INSERTS	REQP	DM	CE
29	NO MORE GENERATED INSERTS	DM	RECP	BC
30	REQUEST ID OF A FINISHED REQUEST	30 RECP	30 CC	30 B
31	AN UPDATE REQUEST HAS FINISHED	RECP	DM	B
31	AN UPDATE REQUEST HAS FINISHED	DM	CC	B

Figure 4.7 MDBS Message Types.

SOURCE OR DESTINATION DESIGNATION		PATH DESIGNATION	
HOST	: HOST MACHINE (TEST-INT)	H	: HOST
REQP	: REQUEST PREPARATION	C	: CONTROLLER
IIG	: INSERT INFORMATION GENERATION	C	: CONTROLLER
PP	: POST PROCESSING	C	: CONTROLLER
DM	: DIRECTORY MANAGEMENT	B	: A BACKEND
RECP	: RECORD PROCESSING	B	: A BACKEND
CC	: CONCURRENCY CONTROL	B	: A BACKEND

Figure 4.8 MDBS Message Abbreviations.

Communication between computers in MDBS is achieved by using a time-division-multiplexed bus called the parallel communication link (PCL) [Ref. 14]. MDBS contains a software interface to this bus for each computer consisting of two complimentary processes. The first process, get-pcl, gets messages from other computers off the PCL. The second process, put-pcl, puts messages on the bus to be sent to other computers. The controller and each backend have their own get-pcl and put-pcl processes.

In the remainder of this section, we give short descriptions of the definitions of MDBS messages. These definitions are of the form:

(message-type number) message-type name: explanation of message.

The descriptions will be given by the process that receives the message. These descriptions are in following figures: Request Preparation (Figure 4.9), Post Processing (Figure 4.10), Directory Management (Figure 4.11), Record Processing (Figure 4.12), Concurrency Control (Figure 4.13), Host processed for Test Interface (Figure 4.14), and Insert Information Generation (Figure 4.15).

(1) Host Traffic Unit : The traffic unit represents a single request or transaction from a user at the host machine.

(13) Record that has Changed Cluster : This message is a record which has changed cluster, Request Preparation will prepare it as an insertion and send it to the backends.

(29) No More Generated Inserts : This message indicates that all the records that have changed cluster as a result of an update request have been sent to Request Preparation.

(14) Results of a Fetch or Retrieve Caused by an Update: This message carries the information from a fetch or retrieve back to Request Preparation to complete an update with a type-III or a type-IV modifier.

Figure 4.9 Request Preparation Messages.

(3) Number of Requests in a Transaction : Request Preparation sends to Post Processing the number of requests in a traffic unit. This enables Post Processing to determine whether the processing of a traffic unit is complete.

(4) Aggregate Operators : Request Preparation sends the aggregate operators to Post Processing.

(5) Requests with Errors : Requests with errors will be found in Request Preparation by the Parser and sent to Post Processing directly. Post Processing will send requests with errors back to the host.

(11) Results of a Request from a Backend : This message contains the results that a specific backend found for a request.

(12) Aggregate Operator Results from a Backend : When an aggregate operation needs to be done on the retrieved records, each backend will do as much aggregation as possible in the aggregate operation function of Record Processing. This message carries those results to Post Processing.

Figure 4.10 Post Processing Messages.

(6) Parsed Traffic Unit : This is the prepared traffic unit sent by Request Preparation.

(29) No More Generated Inserts : This message indicates that insert request for all the records that have changed cluster as a result of an update request have been generated and sent to Directory Management.

(7) New Descriptor Id : This message is a response to the Directory Management request for a new descriptor id.

(8) Backend Number : This message is used to specify which backend is to insert a record.

(15) Descriptor Ids : This message contains the results of descriptor search by Directory Management.

(19) Old and New Values of Attribute being Modified : Record Processing uses this message to check whether a record that has been updated has changed cluster.

(31) An Update Request has Finished : Record Processing signals Directory Management that an update request has finished execution.

Figure 4.11 Directory Management Messages.

(16) Request and Disk Addresses: This message contains a request and disk addresses for Record Processing to come up with the results for the request.

(17) Changed Cluster Response: Directory Management uses this message to tell Record Processing whether an updated record has changed cluster.

(29) No More Generated Inserts : This message indicates that all insert requests generated as a result of an update request have been sent to Record Processing.

(18) Fetch : Fetch is a special retrieval of information for Request Preparation due to an update request with type-IV modifier.

Figure 4.12 Record Processing Messages.

(20) Type-C Attributes for a Traffic Unit : Concurrency Control takes the attributes in this message and determines when Descriptor Search for an attribute can be performed.

(21) Descriptor-id Groups for a Traffic Unit : Concurrency Control takes the descriptor-id groups in this message and determines when Cluster Search for a request can be performed.

(22) Cluster Ids for a Traffic Unit : Concurrency Control takes the cluster ids in this message and determines when a request can continue with Address Generation and the rest of request execution.

(23) Release Attribute : Directory Management uses this message to signal Concurrency Control that a request has performed Descriptor Search on an attribute, and the lock on the attribute held by the request can be released.

(24) Release All the Attributes for an Insert: Directory Management uses this message to signal Concurrency Control that an insert request has performed Descriptor Search on all the attributes, and the locks on the attributes held by the request can be released.

(25) Release Descriptor-Id Groups : Directory Management uses this message to signal Concurrency Control that an insert request has performed Cluster Search for a request, and the locks on the descriptor-id groups held by the request can be released.

(31) An Update Request Has Finished : Directory Management uses this message to signal Concurrency Control that an update request has finished execution, and all the locks held by the request can be released.

(26) Attribute Locked : Concurrency Control signals Directory Management that an attribute is locked for a request, and Descriptor Search can be performed.

(27) Descriptor-Id Groups Locked : Concurrency Control signals Directory Management that the Descriptor-id groups needed by a request are locked, and Cluster Search can be performed.

(28) Cluster Ids Locked : Concurrency Control signals Directory Management that the cluster ids needed by a request can continue with address Generation and the rest of request execution.

(29) Request Id of a Finished Request : Record Processing signals Concurrency Control that a non-update request has finished execution, and the locks on cluster ids held by the request can be released.

Figure 4.13 Concurrency Control Messages.

(2) Request Results : Contains the results for a request after being collected from all the backends and aggregated, if necessary.

Figure 4.14 Host Messages.

(9) Cluster Id : Directory Management sends a cluster id to Insert Information Generation for an insert request. IIG will decide where to do the insert.

(10) Request for New Descriptor Id : When Directory Management has found a new descriptor, it is sent to Insert Information Generation to generate an id.

Figure 4.15 Insert Information Generation Messages.

E. THE EXECUTION OF A RETRIEVE REQUEST

In this section, we describe the sequence of actions for a retrieve request as it moves through MDBS. The sequence of actions will be described in terms of the messages passed between the MDBS processes: Request Preparation (REQP), Insert Information Generation (IIG), Post Processing (PP), Directory Management (DM), Record Processing (RECP) and Concurrency Control (CC). For completeness, we describe the actions which require data aggregation.

First the retrieve request comes to REQP from the host. In the present implementation, it comes from the controller. REQP sends two messages to PP: the number of requests in the transaction and the aggregate operator of the request. The third message sent by REQP is the parsed traffic unit which

goes to DM in the backends. DM sends the type-C attributes needed by the request to CC. Since type-C attributes may create new type-C sub-descriptors, the type-C attributes must be locked by CC. Once an attribute is locked and descriptor search can be performed, CC signals DM. DM will then perform Descriptor Search on m/n predicates, where m is the number of predicates specified in the query, and n is the number of backends. DM then signals CC to release the lock on that attribute. DM will broadcast the descriptor ids for the request to the other backends. DM now sends the descriptor-id groups for the retrieve request to CC. A descriptor-id group is a collection of descriptor ids which define a set of clusters needed by the request. Descriptor-id groups are locked by CC, since a descriptor-id group may define a new cluster. Once the descriptor-id groups are locked and Cluster Search can be performed, CC signals DM. DM will then perform Cluster Search and signal CC to release the locks on the descriptor-id groups. Next, DM will send the cluster ids for the retrieval to CC. CC locks cluster-ids, since a new address may be specified for an existing cluster. Once the cluster ids are locked, and the request can proceed with Address Generation and the rest of the request execution, CC signals DM. DM will then perform Address Generation and send the retrieve request and the addresses to RECP. Once the retrieval has executed properly, RECP will tell CC that the request is done and the locks on the cluster ids can be released. The retrieval results are aggregated by each backend and forwarded to PP. PP completes the aggregation after it has received the partial results from every backend. When PP is done, the final results will be sent to the user.

IV. AN APPLICATION OF THE METHODOLOGIES TO MDBS

In the previous chapters we discussed the separate topics of methodologies for doing internal and external performance measurements of database systems and the Multi-backend Database System (MDBS). This chapter presents the application of these methodologies to MDBS. The initial discussion concerns modification to the MDBS software. We discuss the decisions made during implementation, modification of the user interface process, the backend processes and the controller processes, and the issues resolved during implementation. The next discussion centers on the modifications of the MDBS test environment, which includes test environment changes and software tools. The final discussion identifies measurement programs that were implemented outside of the MDBS environment.

A. THE MODIFICATION OF THE MDBS SOFTWARE

In this section, we begin by presenting the decisions made concerning the implementation of internal and external performance measurements on MDBS. Next, we discuss the modifications of the user interface and the individual MDBS processes. We conclude this section by relating issues which are resolved during the implementation of the performance measurement methodology.

1. Implementation Decisions

When designing and specifying internal and external performance measurement methodologies, decisions must be made as to the most advantageous positions to place the checkpoints, data collections and data aggregations. These

decisions are based on the need to minimize system overhead, and to provide the appropriate level of detail of the test data obtained. Primitives and data structures must be developed which will allow the measurement programs to remain extensible and which are compatible with existing system software. A user interface must be developed which is easy to use, should not require the user to possess any special knowledge of the interface in order to use it and should maintain data in machine readable form which will allow for future expansion of the performance measurement system.

The following implementation decisions are within the bounds of two constraints placed upon us by the current implementation of MDBS along with two constraints we placed on ourselves. The first constraint concerns the virtual memory available to the processes resident on the backends. The operating system on the PDP-11/44 allocates a virtual memory of 64 Kbytes. Each of the MDBS backend processes must fit into a virtual memory of this size. The additional software added as a result of performance measurement has to be constructed so that it will fit in a the very limited memory space remaining in each backend process. The second constraint concerns the initial MDBS design requirements which called for a broadcast bus between minicomputers. Currently a Parallel Communications Link (PCL) is being employed as the inter-computer message-passing mechanism. Messages passed over the PCL are sequentially transmitted from the sender to the receiver. This difference in operation between the PCL and the broadcast bus must be taken into account in our attempt to validate the claims of MDBS. Additional performance measurement programs must also be written to measure message-passing times on the PCL.

The third constraint, i.e., minimizing overhead, significantly influences our performance measurement design. This subject will be discussed in the following paragraphs.

The final constraint deals with our desire to run MDBS unimpeded by the new performance measurement software. When we are not evaluating the system, we want to be able to run MDBS with no overhead incurred by the additional programs and checkpoints of the performance measurement system. This is accomplished by bracketing all performance measurement software within special preprocessor instructions which allow us to include or omit the performance measurement software during program compilation. A definition file is created containing flags which are used to determine the sections of performance measurement code to be compiled. By compiling separate versions, we then have the capability of running MDBS without performance measurement overhead or with the overhead introduced when we select certain portions of the performance measurement software for compilation.

Communication in MDBS is accomplished by passing messages. Processes which are resident in the same minicomputer communicate by using inter-process messages, while processes resident in different minicomputers communicate by using inter-computer messages. Actions taken by the various processes in MDBS are initiated by the receipt of a message. Actions end when that message has been processed and any resultant messages have been sent. As a message is received by a process, the action taken by the process is dependent on the message origination and type. The general MDBS process procedure hierarchy is shown in Figure 5.1.

The highest level of this process is the main procedure. This procedure receives the next message and based upon the originator of the message, calls a sub procedure in the procedure hierarchy. The message works its way down this tree of sub procedures based upon the originator of the message and the message type. Ultimately, the message arrives at a message-handling procedure (message handler). The message handler has the responsibility of processing the

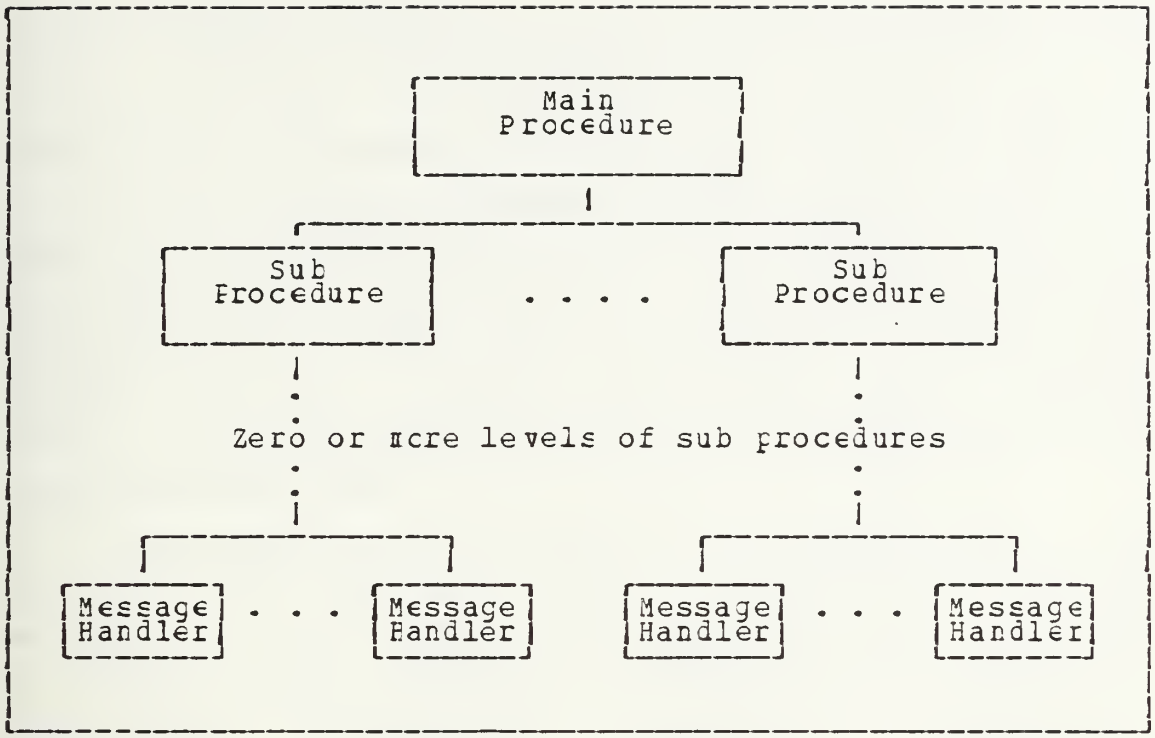

Figure 5.1 The MDBS Procedure Hierarchy.

message. In doing so, it may call other procedures lower in the hierarchy. MDBS's message oriented approach naturally lends itself to checkpoint placement at this level. Selection of measurement at this level provides the user with sufficient processing details while not overburdening the user with excessive information. A range of six to twelve checkpoints may be installed in each MDBS process at this level. The general approach to the installation of checkpoints is shown in figure 5.2. In this installation, we insert checkpoints both before and after every message handler. As a result, we obtain the time of entry into the procedure and exit from the procedure. The differences between these times is the time it takes to process the message.

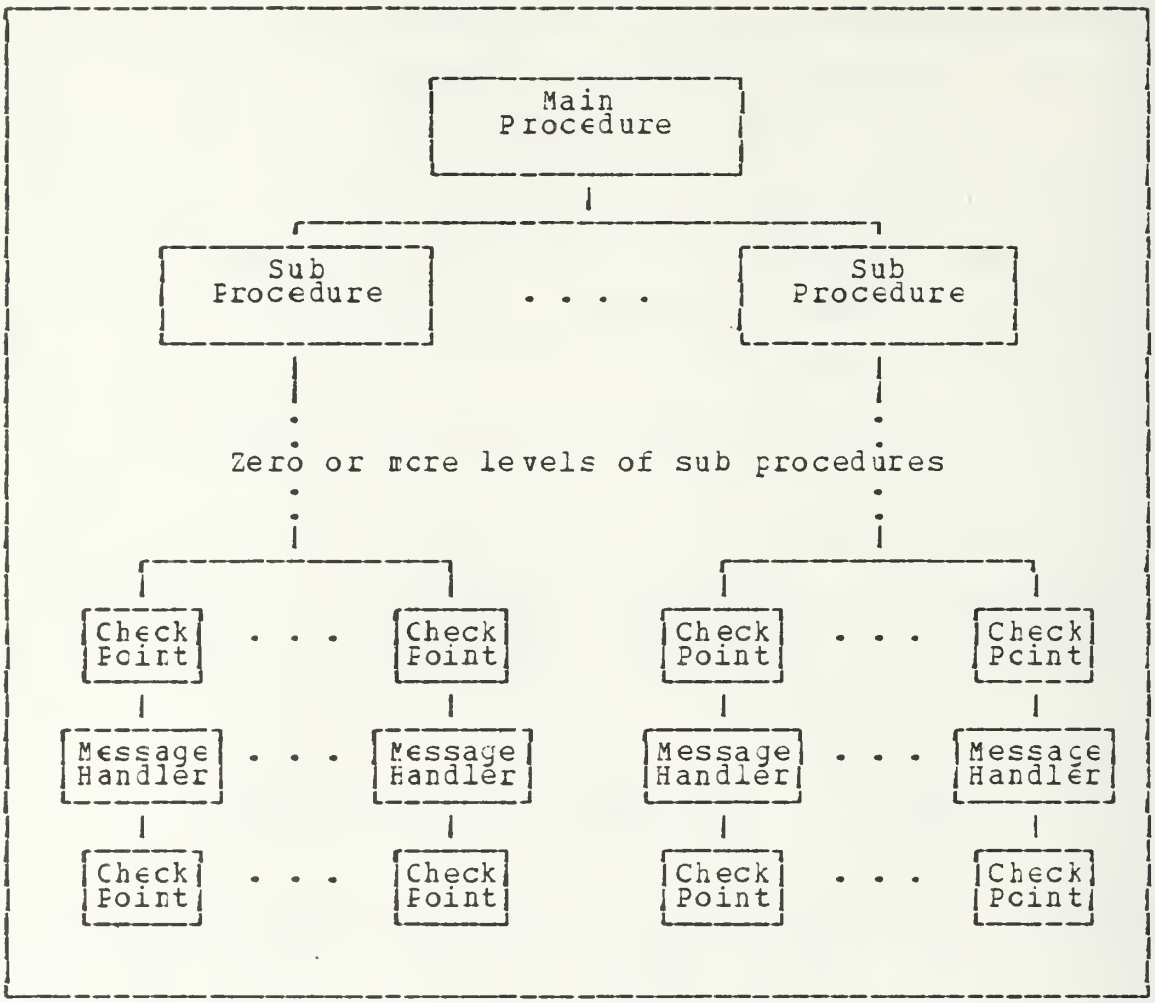

Figure 5.2 The MDBS Procedure Hierarchy with Checkpoints.

Measuring at this level presents one problem. The system clocks are not sufficiently refined for the processing speed of the message-handling routines. The clock on the PDP-11/44 measures time in discrete time intervals of only one sixtieth of a second. The clock on the Vax-11/780 measures time in discrete time intervals of only one hundredth of a second. In any given time interval, the system time may be accessed by the performance measurement software. This means that access may occur exactly when a time interval is recorded by the system clock or anywhere in

between the recording of a time interval by the system clock. Because of this condition, the variance of the time measurement would be approximately twice the smallest interval. This variance is significant when it is compared to the time it takes a message-handling routine to process a message. A method must be developed to reduce this variance. The solution is to send multiple requests to the message-handling routine being timed, to record the time for each request and then to compute the average of the recorded times, thereby obtaining a more accurate measurement of the true processing time.

In order to keep overhead to a minimum and to keep the performance measurement system extensible and simple, we decide to place minimal performance measurement software in an MDBS process. No processing of test data is done in an MDBS process. All test data is sent to the test-interface routines for aggregation and storage. Since MDBS is a message-based system, measurement control messages and test data are transferred as messages utilizing existing MDBS communications routines. A differently-oriented system, such as procedure-oriented, would require a different approach to measurement software communication.

The installation of the checkpoints requires that a method be devised to collect the information obtained by the checkpoints. The information could be stored locally, transferred to a central storage location in the minicomputer or sent to the test interface for storage. In order to reduce the system overhead introduced by message passing we determine that the temporary local storage of data would be most efficient. As pointed out previously, one of the constraints placed upon the implementation of performance measurement is the virtual memory space available at the backends. Storage of the test data generated from the checkpoints would have to be large enough to contain sufficient timing information

and small enough to reside in the constrained virtual memory space available to a backend process. For our timing measurements, the upper bound on the number of requests sent to a message handler at one time is fixed at one hundred. In other words, we assume that measuring a given function more than 100 times will not provide a statistically significant difference over measuring that same function exactly 100 times. Given this upper bound, we decide that a static array of 200 records would be small enough to fit in the virtual memory of a backend process, yet large enough to hold a sufficient amount of test data. Figure 5.3 shows the general approach to the placement of the performance measurement routine (Timer) which is called by the checkpoint, accesses the system clock and manages the static array.

Another question that must be answered is the manner in which the checkpoints are activated. Should we activate only one checkpoint at a time or multiple checkpoints at once? We determine that activating more than one checkpoint at a time could introduce error into the measurement. If one routine (A) which is being measured called another routine (B) which is also being measured, the time necessary to do timing measurements on (B) would increase the total running time of (A). Because of this we only allow the measurement of one routine at a time.

The desire to provide a user interface which is easy to use and requires no particular knowledge of test interface implementation leads us to develop a menu-driven system. The modularity of the performance measurement design lends itself to easy access via menus. The menu-driven system is also compatible with the existing test interface system.

The final problem is how to process and store the raw test data received from the various processes. We require that the user have access to both raw data and

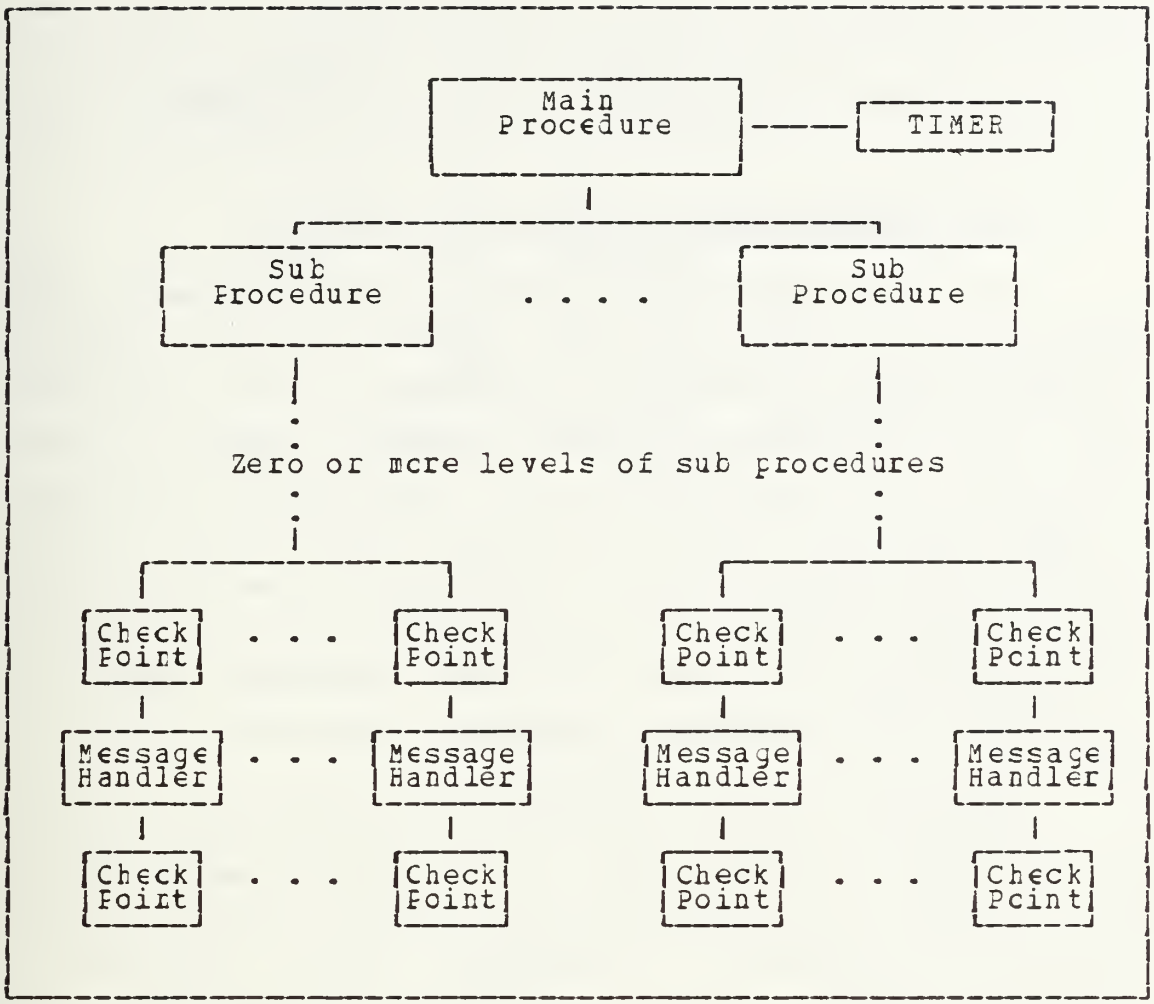

Figure 5.3 The Procedure Hierarchy with Checkpoints and Timer.

summarized information. Also, we require that the data be available for further machine processing. These problems are eliminated by maintaining all collected data in files. When raw data is received from a process it is immediately stored in a file. Once all requests which are to be timed have finished, the file containing the raw data is accessed and processed to produce another file containing summarized information on the various message-handling routines which have been measured. A history of this information is

compiled as the underlying operating system (on the minicomputer where the controller software resides) creates new versions of these files each time the measurement programs are invoked.

2. The Modifications of the User Interface

Prior to the implementation of the performance measurement methodologies, the test interface process consisted of those programs necessary to generate a test database, load a test database and execute requests against the test database. The implementation of performance measurement software within the existing software structure of the test interface is accomplished by expanding the existing hierarchy of control and by integrating performance measurement software with existing test interface software. Figure 5.4 shows the test interface procedural hierarchy with the performance measurement modifications.

The user selects actions to be performed by traversing a tree. At each node, a decision is made as to the path to follow. By following certain paths, the user has the capability to generate a database, load a database or execute the test interface. When the user decides to execute the test interface, a decision is then made as to what path to follow on the test interface sub-tree. The user may choose a new database to work with, create a new list of traffic units, modify an existing list of traffic units, select traffic units from an existing list for execution, select an existing list so that all traffic units on the list may be executed, display the results of external measurement or perform a combination of internal and external performance measurement. The user may traverse the tree at will moving up and down the branches to accomplish a wide variety of tasks.

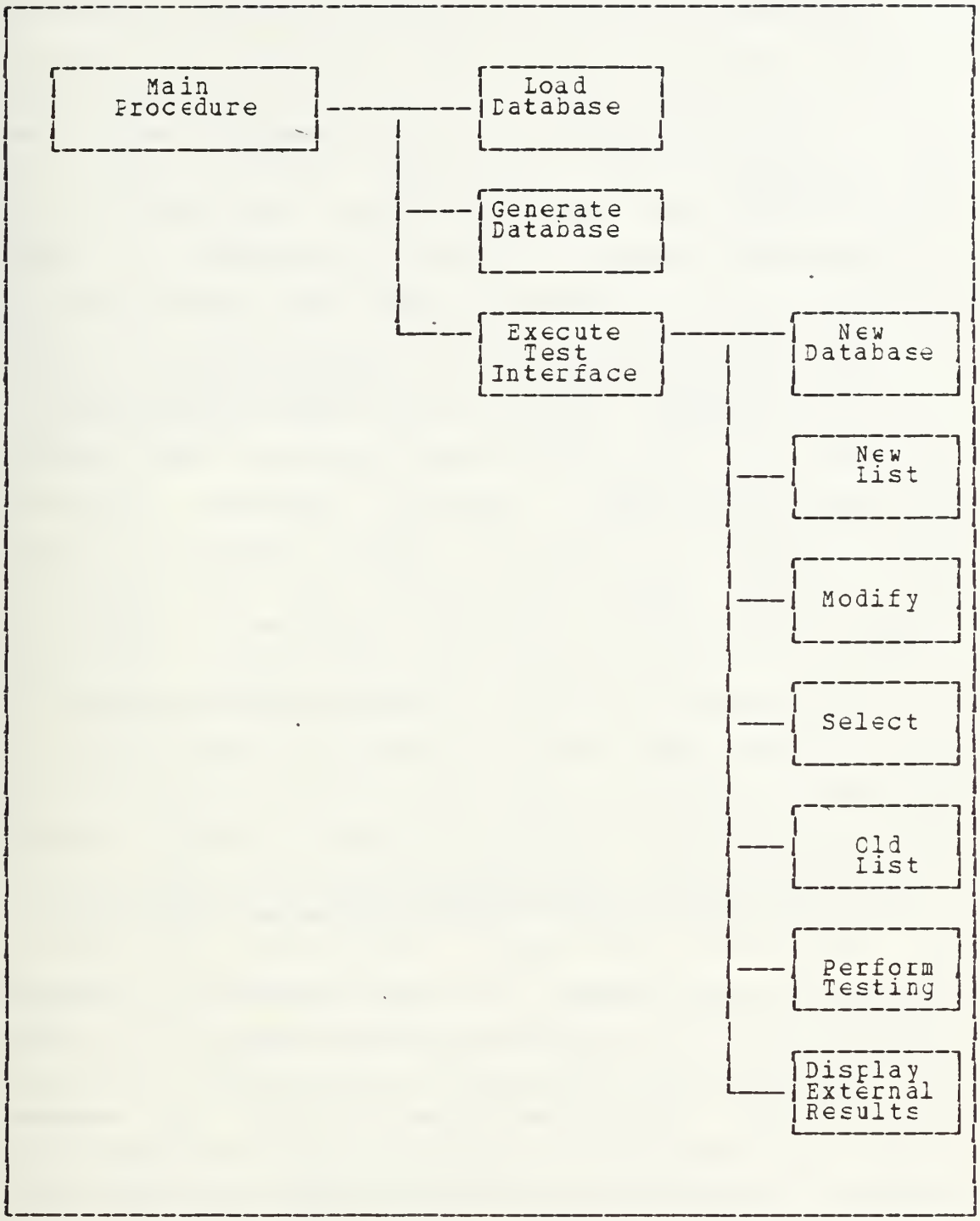

Figure 5.4 The Test Interface Hierarchy with Perfcrmance Measurement Software.

The MDBS software in the test interface contains a procedure, called by the other MDBS procedures, to execute a request (transaction). That is, to forward a request (transaction) to Request Preparation for processing. This procedure is selected for the placement of the external and internal performance measuring software necessary to time and manipulate requests. External measurements are taken from this procedure immediately before the request is sent to Request Preparation for processing and after the results are returned from Post Processing. Software is added to this procedure to generate requests to the MDBS processes which initialize the message-handling routines for internal performance measurement, generate multiple, identical requests in order to reduce the timing variance (as previously discussed) and to generate the test data collection message. The number of multiple requests to generate is provided to this routine by a variable defined at compile time. This procedure receives the information necessary to accomplish these other tasks by sharing a first-in-last-out stack and a pointer to the top of the stack with the performance measurement software. The evaluator interacts with the performance measurement software to build a stack of internal performance measurement requests. This procedure then draws from that stack, initializes the message-handling routine selected by the evaluator, generates multiple copies of the MDBS request selected by the evaluator, and generates the request necessary to collect the test data from the process which contains the message-handler being evaluated. Figure 5.5 shows the relationship between this procedure and the performance measurement software and its data structures.

In addition to external system timing, other performance measurement functions provided by the new software include routines which allow the user to 1) select

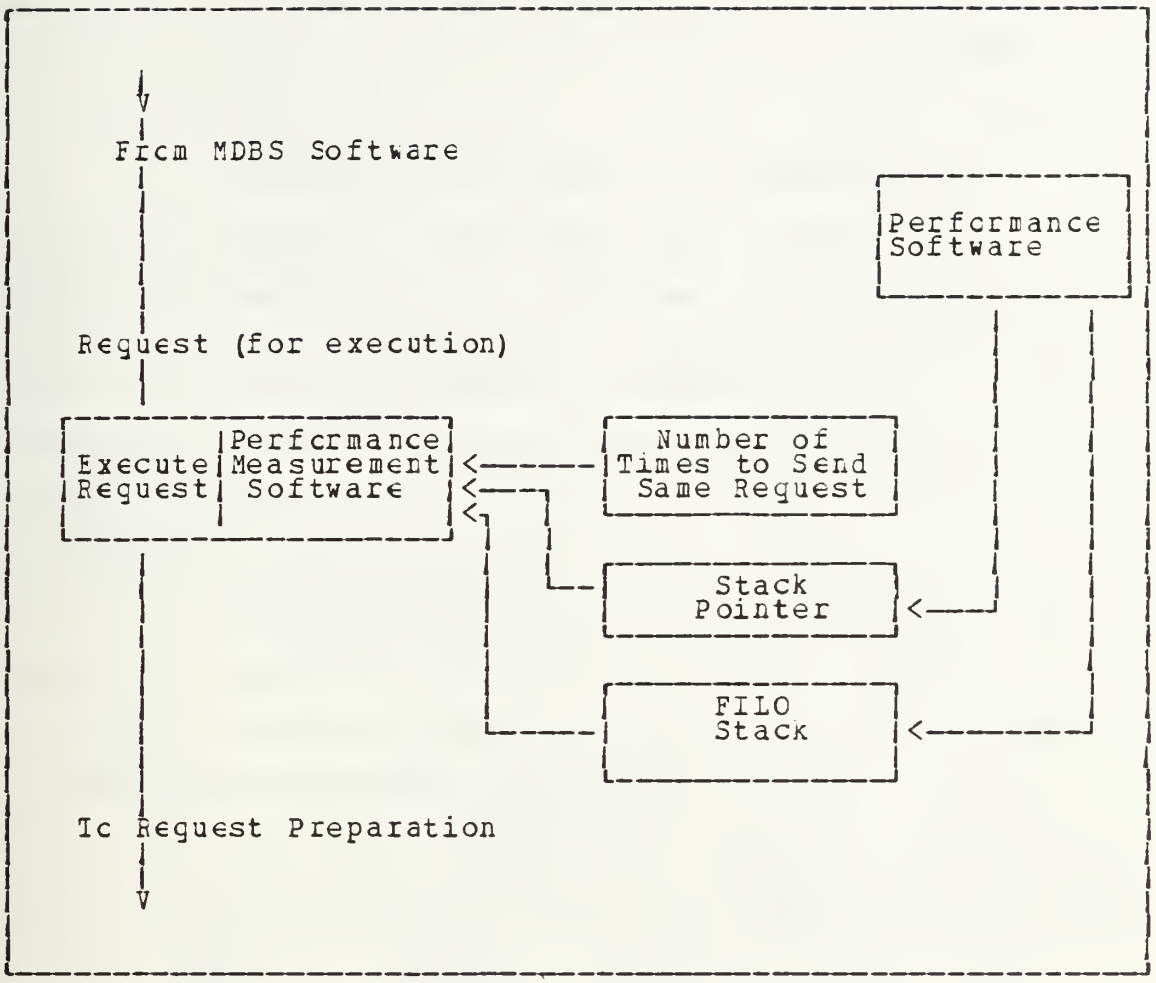

Figure 5.5 The Relationship of the Request Execution and the Performance Measurement Interface.

specific MDBS message-handling routines to be timed, 2) select all message-handling routines within a process to be timed, 3) restrict the timing of backend message-handling routines to a specific backend or backends and 4) perform any combination of the aforementioned selections. The new performance measurement software also includes routines to control the timing software within the MDBS processes, collect raw data transferred to the test interface from processes within MDBS, process the raw data into summarized form and store the data for future use. Other routines are

introduced into existing test interface software to aid in the message-passing requirements of the performance measurement system.

3. The Modification of Individual Processes

The PCL processes within MDBS are modified to pass performance measurement messages. All of the remaining MDBS processes received identical modification. The send/receive portion of every process is modified to include the capability of processing performance measurement messages. Send/receive is used for inter-process message passing. Checkpoints are placed in the MDBS processes at the message-handling (low) routine level. A timer routine is placed in each process which receives control messages from the test interface. An initialization message causes the timer routine to initialize the data collection array to zero and turn on a selected checkpoint. As MDBS-generated messages pass through a checkpoint, the timer routine is called. The timer routine accesses the system clock and stores the message type and time in an array. A completion message from the test interface causes the routine to transmit the data collected in the array to the test interface and to turn off the checkpoint which is timed. Figure 5.6 shows the modifications made to the directory management process as an example of the implementation of the general modifications, shown in figure 5.3.

4. Issues Resolved During the Implementation

MDBS is an experimental database machine. As such, it is under constant modification and subject to use by many system developers. The MDBS software engineering environment requires that versions be used to control program modification, but it is impractical to create new versions of MDBS every time a single program is modified. One solution we

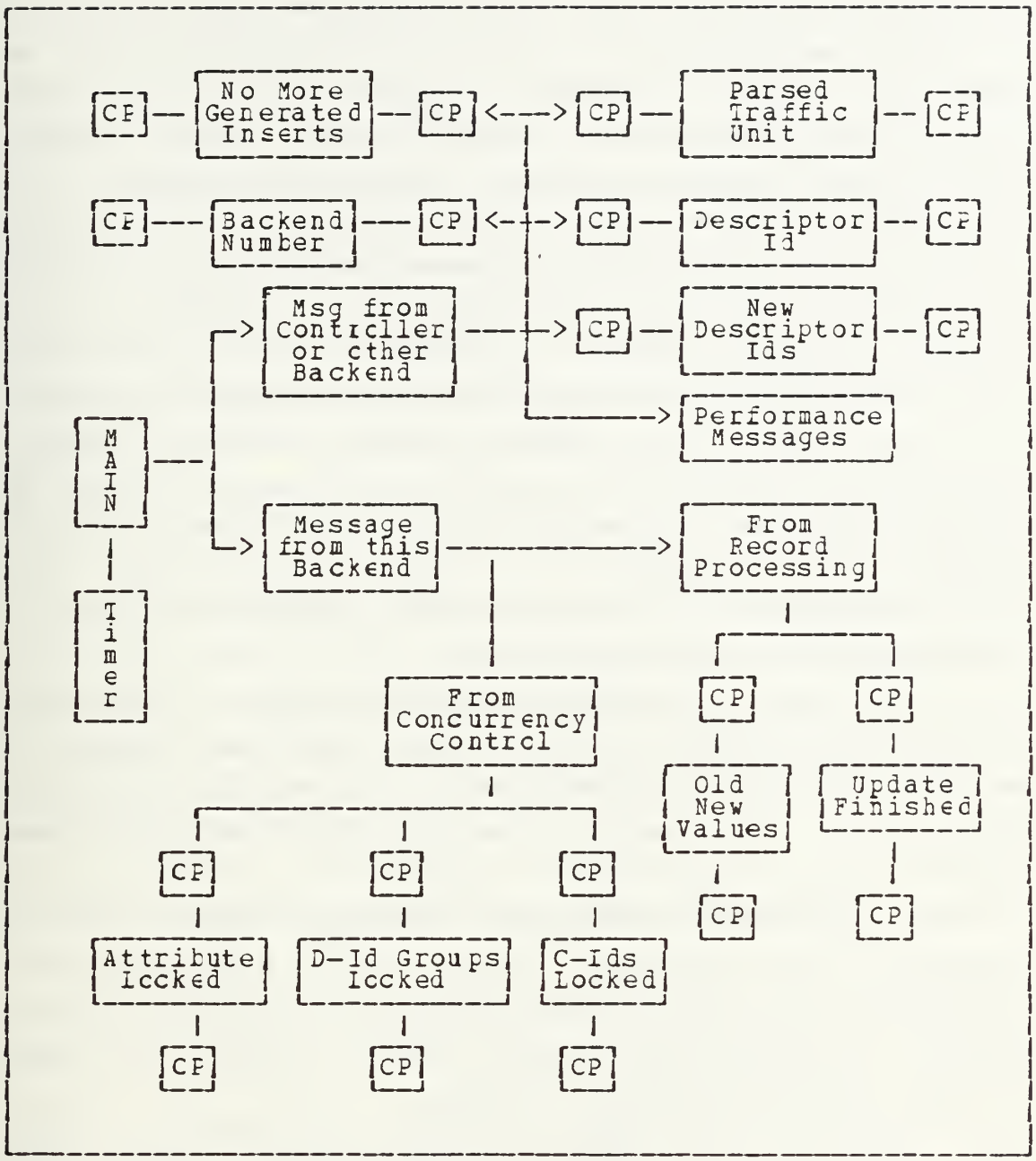

Figure 5.6 The Directory Management Hierarchy with Checkpoints/Software.

implemented is the maintainance of an in-use file. When someone desired to modify a program, the program is copied to the developer's private work space. The developer makes

an entry in the in-use file which indicates who is currently modifying that particular program. This method allowed the developer to modify a program, compile and test the modification in an environment away from the main MDBS environment (in order to avoid compromising a functional system) and return the program to MDBS upon completion of testing. This method avoids the possibility of two developers concurrently modifying the same program and the ensuing problems. Machine time is also at a premium. There is no easy solution for this. Much of the measurement must be conducted during non-peak hours such as late evening and weekends. This is necessitated by the requirement that the measurement of MDBS be accomplished in a stand-alone environment. Since the MDBS controller is implemented on a time-sharing system, the entire machine has to be reserved for performance testing so that MDBS could be run in isolation.

The performance measurement system places additional demands on MDBS system message-passing software. Except for one case, the system responds without protest to this unexpected load. The message-processing routines of the MDBS backends are not designed to handle the transfer of 200 internal performance-measurement messages from a backend process to the controller. There is not sufficient space available to store the pointers required to access this many messages. The MDBS programs are easily extended to account for this change in message traffic.

The MDBS controller resides on a VAX-11/780 which operates under a time-sharing mode. When inter-computer messages are passed on the PCL, the operating system expects a confirmation within a certain time interval. While no problems occur during the normal operation of MDBS, the large message traffic from the backends to the controller during internal measurement require more time than that alloted to the controller during its quantum on the

VAX-11/780. The result is that the controller processes on the VAX are suspended while the backend is still transmiting over the PCL. When the PCL receives no response it signals an error and aborts. Obviously, this is not a problem when the MDBS system runs stand alone. However, such abortion does provide more than an inconvenience during the implementation of the performance measurement system.

Currently, MDBS utilizes two different type of minicomputers. This translates into two different operating systems, two different text editors, two different compilers and two different system clocks which record times in differing units. Because of this, performance programs in the controller processes and the backend processes are not identical. Different access mechanisms for system timers must be developed and a routine must be developed to convert the times received from the backends into the equivalent time units of the controller. Additional time and effort are required to become sufficiently knowledgeable on the two systems in order to begin implementation of the performance measurement methodology.

B. THE MODIFICATION OF THE MDBS TEST ENVIRONMENT

In conducting performance measurements, one demands that all the measurements be consistent as well as reproducible. There should be no inconsistent, unexplainable results. Further, the results should be reproducible with re-runs. This section discusses the necessary changes in the test environment to insure consistency and reproducibility. Then we present the software tools used to make the testing easier and smoother.

1. Necessary Changes to the Test Environment

The methodologies for internal and external performance measurements on a database system have one prerequisite. The results must not be accidental. These results need to be consistent and reproducible. To achieve consistency and reproducibility, we must be able to control the test environment. Every scientific experiment requires the test environment to be controlled, to insure that all factors effecting the experiment are known.

The experimental MDBS, the system to be tested, has its controller processes running under a VAX/VMS environment. This requires these processes of the controller to be run simultaneously with the other non-system processes in a timesharing environment. Under this environment, the results obtained would be erratic and inconsistent. To alleviate this, several preliminary steps are taken prior to final testing. The tests are run stand-alone with all other logins to the computer disabled. All processes are given the highest possible real-time priority. Swapping out of processes in the wait state is disabled to retain the processes in the physical memory. Page faults are disabled by increasing the working set size to the size of the image of each process. In this way, the VAX/VMS system appears to the evaluator as a single user system.

2. Software Tools for the Test Environment

An evaluator should understand the system to be tested, determine the various parameters to be altered, specify the various data to be collected, and interpret the results. Tedious and busy work, such as modifying the input set or the system configuration, can be done manually and are time-consuming without proper tools. Nevertheless, these modifications are necessary, and can be automated by using software tools.

In [Ref. 9], software has been provided to generate a database and a request set, load the database, and run the request set against the database which can all be used in the testing of MDBS. This allows for easy creation and modification of the selected database and requests. The system software needs to be modified during the testing to accommodate such things as changes in the number of backends being used by the system and whether or not internal testing is to be performed. Each change requires a recompilation of the system software. To facilitate this change and to insure only recompilation of necessary files, the Unix 'make' command is used. Briefly, execution of this command would check a file created by the author. This file would indicate all interdependencies of all files of MDBS. If a file has been changed, all other files effected by this change will automatically be recompiled and relinked upon execution of the 'make' command. In this manner, the system could be reconfigured with ease and with the assurance that all effected files are changed. Using these software tools for the test environment and with proper control of the test environment, the tests are made easier to conduct and control and are known to be consistent and reproducible.

C. ADDITIONAL MEASUREMENT SOFTWARE REQUIREMENTS

In order to completely evaluate MDBS, the message passing mechanisms must be monitored to determine the time required to pass both inter-computer and inter-process messages. Although the measurement of these messages could occur during the execution of MDBS, the environment under which the messages are passed could be more easily controlled if the messages are evaluated outside of the MDBS environment. The results of these measurements are contained in the next Chapter VI.

1. Inter-computer Message Processing Measurement

New software does not have to be developed to measure the time required to pass messages on the PCL. Programs are provided by the manufacturer of the PCL which measure the message-passing time. The evaluator is given the capability of specifying which node on the PCL is to receive the message, the message length and the number of messages to send. The software generates and sends the messages, then provides the total time to transmit the messages to the evaluator. The PCL is implemented as a ring bus. Because of this style of implementation, we decide to send messages from one selected node to itself. The times obtained are an upper bound to the inter-computer message passing time.

2. Inter-process Message Processing Measurement

Programs are written for the inter-process message processing measurement. To determine the time required to pass a message, we developed two programs. The first program gets the time, generates a selected number of messages with a selected message length, and sends them to a second program which receives the messages and then gets the time. We run the first program at a higher system priority than the second to prevent the system from process switching before all the messages have been generated. After generation of all messages by the first program, we then set the system priority of the sending program below that of the receiving program, thereby forcing a process switch. We can then compute the average time it takes to pass a single message on the machine. To obtain a higher degree of accuracy we must account for the time it takes the system to switch processes and the time it takes the system to alter the priority of the sending process. Programs are written to account for these times. The program written to account for

the time necessary to alter the priority merely gets the time, alters the priority a selected number of times, then gets the time again. There are two programs necessary to determine the time to process switch. They are identical to the two programs mentioned above except that the number of messages between process switching is set to one. Utilizing the above programs we are able to obtain the inter-process message-passing times on both the PDP-11/44 and the VAX-11/780. The next chapter will discuss the selected database, request sets, and procedures taken to run the actual benchmark of MIBS.

V. THE BENCHMARK OF MDBS

The construction of the test database and the selection of requests are very important in the performance measurement of a test database system. The test database should be representative of a real database, but, as presented in [Ref. 7], the test database should be modeled independent of any specific database. Both the Test database and the requests selected should be properly modeled to allow for a complete exercise of the target system. At the same time, parameters must not be selected randomly, but rather should be created to provide the evaluator flexability and ease of evaluation. In this chapter, we first describe the manner in which the test database is modeled. We then describe the request set which is used in the performance measurement experiments.

A. THE SELECTED DATABASE

Since MDBS is an experimental database system, it is constantly being improved and enhanced. For this reason, the test database is designed to facilitate measurements by being easily expandable. A distinction will be made in the following discussions between the design of the test database, which allows for future measurements, and the actual implementation of the test database used in the measurement experiments.

1. The Design of the Model Database

Several factors must be considered in the design of a model database. Since the system being measured can be configured with either one or two backends, the 'work'

required to process a request has to be evenly divisible to accommodate the use of either one or two backends. The types of work involved are attribute search, descriptor search, cluster search, address generation and the retrieval and selection of records.

Table I displays the three configurations to be used in the performance measurement of MDBS. The configurations have been selected to simplify the verification of the MDBS performance and capacity claims. These claims are to 1) halve the response time by doubling the number of backends and keeping the size of the database constant and 2) maintain a constant response time by doubling both the number of backends and the size of the database. As shown in Table I, going from Test A to Test B maintains a constant database size but allows the database to be evenly split between two backends. Conversely, going from Test B to Test C doubles the size of the database at each backend.

TABLE I

The Benchmark Configuration

Test	No. of Backends	Mbyte/backend
A	1	n
B	2	0.5 n
C	2	n

To properly evaluate a database system, various record sizes need to be used. The sizes are chosen based on the size of the unit of disk management. In MDBS, this is

the logical track, or block. MDBS processes information from the secondary memory using a 4Kbyte logical track. Given a blocksize of 4Kbytes, we recommend constructing the database with record sizes of 200 bytes, 400 bytes, 1000 bytes, and 2000 bytes [Ref. 7]. This gives a range of 2 to 20 records per block. This also creates an environment where four separate databases, corresponding to the four record sizes, must be generated and tested for each configuration given in Table I. Table II gives the corresponding relationship between records and blocks.

TABLE II

The Record-and-Block Relationship

Record Size in Bytes	Records per Block
200	20
400	10
1000	4
2000	2

As described in Chapter III, the target system stores records in clusters. Five cluster categories have been selected for use in the creation of the model database. The distinguishing characteristic of a cluster category is the number of blocks used to store the records in the particular category. Table III outlines the sizes of each of the five cluster categories. One final note, the number of blocks per cluster must be even. Thus, when the number

of backends is increased from one to two with the number of records in the database remaining constant, we are guaranteed that each backend will have the same number of blocks per cluster. For example, when the cluster category is small, each backend would have one block for the particular cluster, insuring an even distribution of the database.

TABLE III

The Cluster Arrangement

Clusters Category	Blocks/Cluster
small	2
small-medium	4
medium	6
medium-large	8
large	10

Combining the data in Tables II and III, we can construct a matrix of data which represents the number of records per cluster category. Table IV, indexed using the cluster category and the record size, details this information. The number of records per cluster is obtained by multiplying the Records/Block results from Table II by the corresponding Blocks/Cluster results from Table III.

The remaining considerations when developing a test database involve the specification of the directory structure for the particular record type. In MDBS, a record template, which describes the record structure is defined. The record template defines the number of attributes in the

TABLE IV

The Records per Cluster Category

CLUSTER CATEGORY (Record Size in Bytes)	Number of Records per Cluster (200)	(400)	(1000)	(2000)
small	40	20	8	4
small-medium	80	40	16	8
medium	120	60	24	12
medium-large	160	80	32	16
large	200	100	40	20

record, and for each attribute, the attribute name and the attribute type (either integer or string). Given a record template, the directory and non-directory attributes are specified. For each directory attribute, a descriptor type and descriptor ranges are defined (see Chapter III).

2. The Implementation of the Model Database

This section examines the implementation decisions made when specifying the test database and the testing strategy. The current version of MDBS, the primary-memory-based directory management, stores the directory tables, i.e., the AT, DDIT, and CDT, in primary memory. Given the primary memory limitations of the backend, we are forced to limit the variables mentioned in the previous section. Our first decision is to limit the size of the test database to a maximum of 1000 records per backend. Table V displays the configurations which are used in the performance measurements of MDBS.

TABLE V

The Measurement Configurations

TEST	No. of Backends	Records/Backend
A.E	1	1000
B.E	2	500
C.E	2	1000
A.I	1	1000
B.I	2	500

Five different system configurations are needed for the MDBS performance measurements. Tests A.E, B.E, and C.E are conducted without internal performance software in place. Test A.E configures MDBS with one backend and one thousand records in the test database. Test B.E configures MDBS with two backends and one thousand records split evenly between the backends. Test C.E also configures MDBS with two backends, but, the size of the database is doubled to two thousand records. Test A.I and B.I are conducted with internal performance software in place. Test A.I configures MDBS with one backend and one thousand records in the test database. Test B.I configures MDBS with two backends and one thousand records split evenly between the backends. Using these five configurations, the verification of the MDBS performance and capacity claims is simplified and the performance measurement methodology of computing the internal measurement overhead is facilitated.

Our second decision fixes the record size at 200 bytes. The 200 byte record minimizes the primary memory required to store the record template. In actuality, a record of 198 bytes is used. The record consists of 33 attributes, each requiring 6 bytes of storage. The record

template file used in our experiments is shown in Figure 6.1. Of the 33 attributes listed, INTE1 and INTE2 are directory attributes. MULTI and STR00 to STR29 are non-directory attributes.

In our next decision, the descriptor types and the descriptor ranges for the two directory attributes, INTE1 and INTE2, are defined in the descriptor files (see Figure 6.2). The values for INTE1 are classified by using five type-A descriptors, each of which represents a range of 200. The values for INTE2 are also classified using type-A descriptors. The first twenty-three ranges for INTE2 cover 40 values, with the last range covering 80 values. The non-uniformity of the INTE2 descriptor ranges is caused by a size constraint in the Concurrency Control process.

Attribute Name	Attribute Type
INTE1	integer
INTE2	integer
MULTI	string
STR00	string
STR01	string
.	.
.	.
.	.
STR29	string

Figure 6.1 The Record Template File.

By utilizing the attribute and descriptor files, the record file is generated. INTE1 and INTE2 are identical, being the next sequential number after the previous record, starting at 1. Therefore, the one thousandth record would have the (INTE1, INTE2) pair set to 1000. The MULTI attribute, which is of type character string, is set to One for a database of only 1000 records. The intent of this attribute

Attribute Name	Descriptor Type	Descriptor Range
INTE1	A	1 -> 200 201 -> 400 401 -> 600 601 -> 800 801 -> 1000
INTE2	A	1 -> 40 41 -> 80 81 -> 120 121 -> 160 161 -> 200 201 -> 240 241 -> 280 281 -> 320 321 -> 360 361 to 400 401 -> 440 441 -> 480 481 -> 520 521 -> 560 561 -> 600 601 -> 640 641 -> 680 681 -> 720 721 -> 760 761 -> 800 801 -> 840 841 -> 880 881 -> 920 921 -> 1000

Figure 6.2 The Descriptor File.

INTE1	INTE2	MULTI	STR00	STR01	...	STR29
1	1	One	Xxxxx	Xxxxx	...	Xxxxx
2	2	One	Xxxxx	Xxxxx	...	Xxxxx
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
1000	1000	One	Xxxxx	Xxxxx	...	Xxxxx

Figure 6.3 The Record File.

is to increase the number of records per cluster in the database. This is done by setting MULTI to Two, Three, etc., for each (INTE1, INTE2) pair in the database. Therefore, to double the size of the database, every (INTE1, INTE2) pair will have an associated MULTI attribute with values of One and Two. The remaining attributes, STR00 to STR29, are set to Xxxxx as fillers. Figure 6.3 depicts the general layout of the record file for 1000 records where MULTI is set to One.

Given the structure described, our last decision is made for us. The specification of 24 descriptors for the INTE2 attribute, coupled with the record file structure, generates a database that contains 24 clusters. The first 23 clusters correspond to the small cluster category, and each contains 40 records. The last cluster corresponds to the small-medium cluster category and contains 80 records. To maintain consistency in the retrieval requests (discussed in the next section), we avoid any requests that access the last 80 records in the test database using the INTE2 attribute.

B. THE REQUEST SET

The request set used for our performance measurement is given in Figure 6.4. The retrievals are a mix of single or double predicate requests. Since the majority of the work done on a database is to retrieve data, we limit the measurements to only retrieve requests. In every request, 1/2 of the target attribute values for each record is returned. The first retrieve is a request for only two records from two separate clusters. The second request retrieves 1/4 of the database. Seven of the 24 clusters must be examined. All records in each of the first six clusters are retrieved. Only 1/4 of the seventh cluster, defined by the range from

Request Number	Retrieval Request
1	(INTE1=10) or (INTE1=230)
2	(INTE2 ≤ 250)
3	(INTE2 ≤ 500)
4	(INTE1 ≤ 1000)
5	(INTE1≤200) or (INTE1≥801)
6	(INTE1≤400) or (INTE1≥601)
7	(INTE1 ≤ 201)
8	(INTE1 ≤ 401)
9	(INTE1≤201) or (INTE1≥800)

The Target Attribute-Values for Each:

(INTE1,INTE2,MULTI,STR00,STR01,STR02,STR03,STR04, STR05,STR06,STR07,STR08,STR09,STR10,STR11,STR12)

Figure 6.4 The Retrieval Requests.

241 to 280, is retrieved. In the third request, 1/2 of the database is retrieved. Thirteen of the 24 clusters must be examined. All records in each of the first twelve clusters are returned. Only 1/2 of the thirteenth cluster, defined by the range from 481 to 520, is retrieved. The system searches only for records having values in the range from 481 to 500 in this cluster.

The entire database is examined in the fourth request. The fifth request retrieves 2/5 of the database. The query is divided into two predicates, to obtain all records from the first five clusters, and the last four clusters. The sixth request is a retrieval of 4/5 of the database. Again the query is divided into two predicates, to obtain all records from the first 10 clusters, and the last nine clusters.

The seventh and eighth requests are similar in intent. The seventh request examines 10 clusters, requiring only 1 record to be retrieved from the 6th cluster and needing all records from the first five clusters. The eighth request examines 15 clusters, requiring only 1 record to be retrieved from the 11th cluster and needing all records from the first ten clusters. The ninth and final request is similar to the fifth request. But unlike the fifth request, ten additional clusters must be examined. Only two of the records with INTE1 values of 201 and 801, are retrieved from the ten additional clusters. All records in the remaining nine clusters, like the fifth request, are also obtained by this retrieval. Table VI, a presentation of the number of clusters examined versus the percent of the database retrieved, is a synopsis of the previous discussion in tabular form.

The request set in Figure 6.4 is not intended to be representative of a comprehensive and complete request set. The goal is not to exhaustively measure and evaluate MDBS. Rather, we focus on applying the performance measurement methodology to MDBS to validate the basic performance and capacity claims of the system. We feel that these requests are sufficient for such a validation. We will refer to these nine requests, i.e., retrievals, by their record number in subsequent discussion.

TABLE VI

The Number of Clusters Examined
and the Percent of the Database Retrieved

Request Number	Number of Clusters Examined	Volume of Database Retrieved
1	2	2 records
2	7	25%
3	13	50%
4	24 (all)	100%
5	9	40%
6	19	80%
7	10	20% + 1 record
8	15	40% + 1 record
9	19	40% + 2 records

VI. THE TEST RESULTS

In this chapter, we present the results obtained from the performance measurement of MDBS. MDBS is currently configured with the primary-memory-based directory management. In this version of MDBS, the directory tables, i.e., the AT, DDIT, and CDT, are stored in the primary memory. We expect to achieve different results when version F, the secondary-memory-based directory management is implemented. The test interface is utilized to send the retrieval requests discussed in the previous chapter to MDBS for processing. Each request is sent a total of ten times per database configuration. The response time of each request is recorded. After some trial runs, we compute the standard deviation. We determine that ten repetitions of each request is sufficient to provide the desired accuracy.

The internal processing times of the message-handling routines which are used to process a retrieval request are also timed. Retrieval (1) and Retrieval (2) are selected to conduct internal timing. These requests are selected since they retrieve the smallest portion of the test database and the processing time for each request is minimal. Recall that each message-handling routine is timed independently of all others and that each routine must process multiple requests so that an accurate average may be computed for the time required to process that request type. Sixteen message-handling routines are required to process a retrieve request. If we send twenty requests to each routine, a total of 320 requests must be processed by MDBS. Based on these figures, the time required to conduct the internal performance measurement of a retrieval that has a response time of twenty seconds will be approximately 107 minutes. This

figure does not include the administrative time required to process the internal measurement data. For this reason, we limited the internal performance measurement requests to Retrievals (1) and (2).

Additionally, we also limited the number of repetitions per message handler to twenty. This is done to reduce the processing time per message handler. However, this decision reduces the accuracy of the internal performance measurement, from ten-thousands to hundredths of a second. Thus, the internal performance measurement times provide only a rough estimate of the time required to handle the respective messages.

The first section of this chapter contains the external timing results obtained from our measurements. We also discuss the performance and capacity improvements obtained by adding backends. In the second section we present the results from internal performance measurement. The final section examines the inter-process and inter-computer message processing times. One final note, the units of measurement presented in the tables of this chapter are expressed in seconds.

A. THE EXTERNAL PERFORMANCE RESULTS

Table VII provides the results of the external performance measurement of MDBS without the internal performance measurement software. There are three parts to Table VII. Each part contains the mean and the standard deviation of the response times for Retrievals (1) through (9), which are outlined in Chapter V. The three parts of Table VII represent three different configurations of the MDBS hardware and the database record capacity. The first part has MDBS configured with one backend and the database loaded with 1000 records. The second part has MDBS configured with two

TABLE VII

The Response Time
Without Internal Performance Evaluation Software

Request Number	One Backend 1K Records		Two Backends 1K Records		Two Backends 2K Records	
	mean	stdev	mean	stdev	mean	stdev
1	3.208	0.0189	2.051	0.0324	3.352	0.0282
2	13.691	0.0255	7.511	0.0339	14.243	0.0185
3	26.492	0.0244	14.164	0.0269	26.737	0.0405
4	52.005	0.0539	26.586	0.0294	52.173	0.0338
5	21.449	0.0336	11.309	0.0375	21.550	0.0237
6	42.235	0.0326	21.622	0.0424	42.287	0.0400
7	12.285	0.0408	6.642	0.0289	12.347	0.0371
8	22.532	0.0296	11.764	0.0300	22.583	0.0110
9	23.913	0.1115	12.624	0.0350	24.169	0.0181

backends, with the database containing 1000 records, split evenly between the backends. The third part has MDBS configured with two backends, with the database doubled to 2000 records, also split evenly between the backends. In Table VII we notice one data anomaly, the standard deviation for request (9) in the one-backend-with-1000-records configuration. Since we did not conduct an internal performance measurement on this request, we are not sure what causes this skewed standard deviation, and hence will not attempt to offer an explanation of this anomaly.

Given the data presented in Table VII, we can now attempt to verify or disprove the two MDBS performance claims. We begin by calculating the response-time improvement of MDBS. The response-time improvement is defined to be

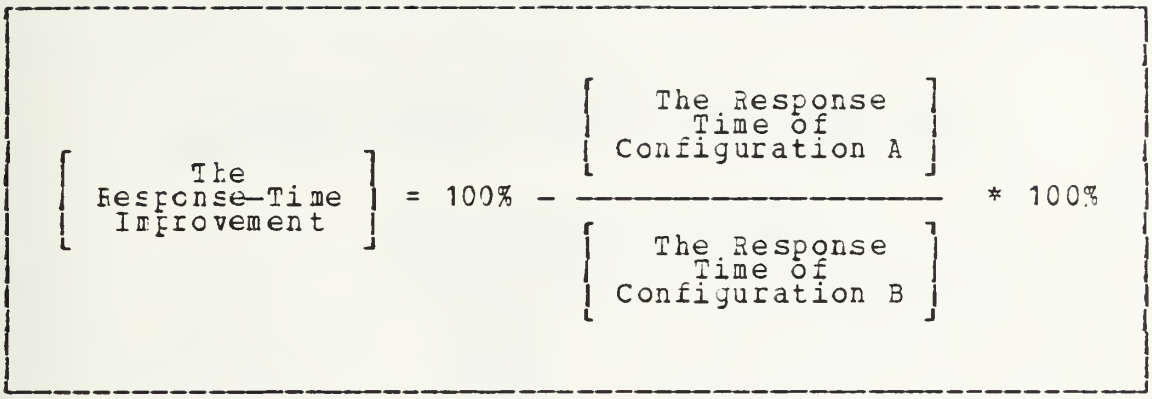

Figure 7.1 The Response-Time-Improvement Calculation.

the percentage improvement in the response time of a request, when the request is executed in n backends as opposed to one backend and the number of records in the database remains the same. Figure 7.1 provides the formula used to calculate the response-time improvement for a particular request, where Configuration B represents n backends and Configuration A represents one backend. Thus, in Table VIII we present the response-time improvement for the data given in Table VII. Notice that the response-time improvement is lowest for request (1), which represents a retrieval of two records of the database. On the other hand, the response-time improvement of request (4), which retrieves all of the database information is highest, approaching the upper bound of fifty percent. In general, we find that the response-time improvement increases as the number of records retrieved increases. This seems to support a hypothesis that even if the database grows, the response-time improvement will remain at a relatively high (between 40 and 50 percent) level.

Next we calculate the response-time reduction of MDBS. The response-time reduction is defined to be the the reduction in response time of a request, when the request is

TABLE VIII

The Response-Time Improvement Between
1 and 2 Backends (External Measurement Only)

Request Number	Response Time Improvement
1	36.07
2	45.14
3	46.53
4	48.94
5	47.27
6	48.81
7	45.93
8	47.79
9	47.21

1K Records
No Internal-
Measurement Software

executed in n backends containing nx number of records as opposed to one backend with x number of records. Figure 7.2 provides the formula used to calculate the the response-time reduction for a particular retrieval request, where configuration A represents one backend with x records and configuration B represents n backends, each with x records. In Table IX we present the response-time reductions for the data given in Table VII. Notice that the response-time reduction is worst for request (1), which represents a retrieval of two records of the database. On the other hand, the response-time reductions for the retrievals which access larger portions of the database, requests (4) and (6), have only a small response-time reduction. In general, we found that the response-time reduction decreases as the number of records retrieved increases, i.e., the response time remains virtually constant. Again we seem to have evidence to support the hypothesis that, as the size of the database

increases, the response-time reduction will decrease to a relatively low (0.1% or less) level.

$$\left[\begin{array}{c}\text{The}\\ \text{Response-Time}\\ \text{Reduction}\end{array}\right] = 100\% * \left[1 - \frac{\left[\begin{array}{c}\text{The Response}\\ \text{Time of}\\ \text{Configuration B}\end{array}\right]}{\left[\begin{array}{c}\text{The Response}\\ \text{Time of}\\ \text{Configuration A}\end{array}\right]}\right]$$

Figure 7.2 The Response-Time-Reduction Calculation.

TABLE IX

The Response-Time Reduction
In Doubling the Database Size

Request Number	Response Time Reduction
1	4.49
2	4.03
3	0.92
4	0.32
5	0.47
6	0.12
7	0.50
8	0.23
9	1.07

1K Records on each Backend
No Internal-Measurement Software

Table X provides the results of external performance measurement of MDBS with internal performance measurement software in place. There are two parts to Table X. Each part contains the mean and the standard deviation of the response-times for the requests (1) through (6), which are outlined in Chapter V. The two parts of Table X represent two different configurations of the MDBS hardware and the database record capacity. Part one has MDBS configured with one backend and the database loaded with 1000 records. Part two has MDBS configured with two backends, with the database containing 1000 records, split evenly between the backends. We did not measure the response times with two thousand records distributed over two backends. We felt that no additional information would be gained by conducting the measurements.

TABLE X

The Response Time (in seconds)
With Internal Performance Measurement Software

Request Number	One Backend 1K Records		Two Backends 1K Records	
	mean	stdev	mean	stdev
1	3.205	0.0436	2.219	0.0474
2	13.418	0.0172	7.401	0.0277
3	25.903	0.0119	13.854	0.0361
4	50.750	0.0374	26.402	0.0596
5	20.972	0.0271	11.244	0.0528
6	41.262	0.0331	21.517	0.0575

An interesting anomaly is discovered when we compare the response times of the external and internal performance measurement tests, i.e., parts one and two of Tables VII and

X for requests (1) through (6). We actually found a general improvement, from 0.1% to 5%, in the response times of the requests when the internal performance measurement software is part of the MDBS code. One hypothesis is that this is due to the manner in which MDBS is implemented on the backends. Currently, there is not sufficient physical memory available on each backend. The result is that disk overlays are used to swap in the code necessary to run MDBS. The additional internal performance measurement code may cause the operating system to overlay differently, thereby benefiting the overall performance of MDBS. We still believe that there is an overhead induced by the internal measurement code and Table XI provides evidence by demonstrating that the response-time improvement achieved by adding a backend is not as good as that of Table VIII.

TABLE XI

The Response Time Improvement Between 1 and 2 Backends (With Internal Measurement Also)

Request Number	Response Time Improvement
1	30.76
2	44.84
3	46.52
4	47.98
5	46.39
6	47.85

1K Records Evaluation Software

B. THE INTERNAL PERFORMANCE RESULTS

Table XII provides the results of the internal performance measurement of MDBS for a retrieval request. The times measured for each message-handling routine are given for both request (1) and (2). The message-handling routines are listed with the MDBS process which contains the routine. Although the results are given to four decimal places, we only trust the accuracy to the second decimal place. The reason for this has been discussed in the introduction to this chapter. We are not experts on the MDBS system. We can, however, make a few comments on Table XII and we are sure that those who are experts can use the results contained in Table XII to draw more in-depth conclusions on the system. We see that the controller processes, i.e., Request Preparation and Post Processing, spend very little time in processing the retrieval request. This is a major design goal of MDBS and is necessary to prevent a bottleneck at the controller when the number of backends increases substantially. It appears that this goal is met successfully. We also observe that the results obtained from Concurrency Control are consistent and of short duration. This is expected since there is only one request in the system at a time and no access contention can occur. These tables should then be considered as containing the best-case times. The majority of work done in the backend is at Record Processing. Observing the process timings in Record Processing, we see that, for both requests, the addition of an extra backend reduces the record processing time by nearly half.

C. THE MESSAGE PROCESSING RESULTS

Table XIII provides some average times relating to inter-process message passing times on the controller and

TABLE XII

Message Handling Routine
Processing Times for a Retrieval Request

MIBS Process	Message Handling Routine	Request Number	One Backend 1K Records	Two Backends 1K Records
Request Preparation	Record Count To Post Proc	1 2	0.0005 0.0000	0.0015 0.0000
	Parse Traffic Unit	1 2	0.0200 0.0180	0.0190 0.0185
	Broadcast Request	1 2	0.0025 0.0065	0.0025 0.0030
Post Processing	Collect Results	1 2	0.0465 0.0890	0.0250 0.0813
Directory Management	Parsed Traffic Unit	1 2	0.0699 0.0925	0.0450 0.0491
	Did Sets Locked	1 2	0.0516 0.0566	0.0566 0.0566
	Cid Sets Locked	1 2	0.0533 0.0450	0.0349 0.0433
	Descriptor Ids	1 2	na na	0.0391 0.0558
Concurrency Control	Cids for Traffic Unit	1 2	0.0424 0.0425	0.0433 0.0433
	Did Sets Traffic Unit	1 2	0.0566 0.0508	0.0408 0.0516
	Did Sets Released	1 2	0.0025 0.0008	0.0016 0.0008
Record Processing	Entire Process	1 2	2.6462 12.7100	1.3775 6.5716
	Request with Disk Address	1 2	0.0466 0.0433	0.0433 0.0383
	Cld Request	1 2	0.0130 0.0131	0.0148 0.0168
	FIO Read	1 2	0.0844 0.8593	0.0865 0.8863
	Disk Input/Output	1 2	0.0799 0.0783	0.0741 0.0725

TABLE XIII

Inter-process Message Passing Times

Location	Time to Construct Message	Time to Receive Message	Time to Pass Message
Controller	0.00249	0.00267	0.00520
Backend	0.00830	0.00410	0.01250

the backend. Messages are transmitted between two processes on both the controller and backend. Both the number of messages and the message length are varied. On the controller, the number of messages is varied from 1 to 100 while the message length is varied from 2 to 2000 bytes (size of the message buffers in MDBS controller). On the backend, the number of messages is varied from 1 to 50 while the message length is varied from 1 to 1000 bytes (size of the message buffer in MDBS backends). It takes the backend twice as long to process a message as it does the controller. We believe the reason to be hardware processor speed. An independent test showed that this relationship, of two to one, holds in how long it takes to process an assignment statement on the respective hardware.

In Table XIV we provide information concerning the time to process inter-computer messages on the PCL. Messages of length less than forty are overshadowed by the overhead of the PCL. There exists a linear relationship between the message length and the time to pass a message as the message length exceeds 100 bytes. We can therefore expect a linear performance from the PCL for the majority of the MDBS inter-computer messages. The next chapter will contain some concluding remarks and discuss areas for further research.

TABLE XIV

Inter-computer Message Passing Times

Message Length (Bytes)	Time to Pass Message	Change
10	0.0949	0.0000
20	0.0951	0.0002
30	0.0954	0.0003
40	0.0957	0.0003
50	0.1005	0.0008
60	0.1011	0.0006
70	0.1018	0.0007
80	0.1023	0.0005
90	0.1029	0.0006
100	0.1036	0.0007
200	0.1136	0.0100
300	0.1238	0.0102
400	0.1339	0.0101
500	0.1439	0.0100
1000	0.1943	0.0504

VII. THE CONCLUSION

A. A SUMMARY OF THE PERFORMANCE MEASUREMENT METHODOLOGY

1. The Internal Performance Measurement Methodology

The internal performance measurement methodology provides the strategies and locations for the placement of checkpoints. It further provides the kinds of performance data to be collected. This information enables a better understanding of the target system by measuring certain capabilities, such as the time spent in individual processes. Using this information of how the system performs internally may lead to design modifications or to fine-tuning of the system for increased performance.

2. The External Performance Measurement Methodology

The external performance measurement methodology provides the strategies for a macro view of the database system performance by measuring the system as a whole. We focus on the measurement of the response time of the target system after the issuance of a request. A test database and a test request set is generated using software tools.

3. Combining the Internal and External Measurement Methodologies

The natural combination of the internal and external performance measurement methodologies is synergistic in the amount of information that is provided. The overhead incurred when using internal performance measurement code is accurately determined using this methodology combination. The external performance measurement timings can be properly interpreted using the internal performance measurement

results. By combining the two measurements, the whole of the measurement results is more meaningful and useful than the individual results.

B. A SUMMARY OF THE METHODOLOGY APPLICATION

Thrilling and unexpected results are collected when this methodology is applied to a target system, i.e., MDBS. First, the methodology proves itself to be successful in attempting to verify the performance and capacity claims of MDBS. This results from being able to collect sufficient data on a target system to make definitive statements concerning its performance. The application of this methodology to MDBS is also surprisingly easy.

A second result, is that the performance and capacity claims of MDBS have been validated. These claims are: 1) that by increasing the number of backends used as a part of the database system and by keeping the size of the database constant, the response time of the same transactions is proportionally decreased, and 2) that by increasing the number of backends and also increasing the size of the database, the response time remains relatively constant. These claims are validated by the results given in Chapter VI.

These spectacular results provide a wealth of information from which several conclusions can be made. We find that under MDBS, the response-time improvement increases as the number of records retrieved increases. Also, the response-time reduction decreases as the number of records retrieved increases. Though the performance measurement results indicate an improvement in the response time of the requests when the internal performance measurement software is part of MDBS code, it is felt that this phenomenon is the result of differing system overlays and that the induced overhead of internal measurement code still needs to be calculated.

The results of the internal performance measurements indicate that the controller processes, i.e., Request Preparation and Post Processing, spend very little time to process the retrieval request. The results obtained from Concurrency Control are both consistent and of short duration, as expected. The results also show that the majority of work is being done in Record Processing and that the addition of a backend reduces the record processing time by nearly half. We discovered that it takes the backend twice as long to process a message as it does the controller, possibly due to hardware processor speed. Finally, there exists a linear relationship between the message length and the time to pass a message as the message length exceeds 100 bytes.

C. RECOMMENDATIONS FOR FUTURE EFFORTS

Future improvements can be made in the performance measurement methodology by the automation of the existing external software tools. Specifically, the ability to start a test which will execute a pre-determined set of requests a pre-determined number of times for each request, and collect the results in a file is a desireable feature. Additionally, since the methodology is intended to be general in use, the methodology needs to be applied to different database systems to discover its applicability, ease of use, and usefulness in overall performance measurement of the target system.

In terms of the application of this methodology to MDBS, a complete and thorough test of the system needs to be conducted. An exhaustive test of MDBS would include conducting test with databases that have varying record sizes. Further, testing the system by varying the number of directory attributes, descriptors, and clusters would indi-

cate the role of the directory data in the system. Insert, delete, and update requests must also be measured to discover their impact on system performance. Lastly, the measurement should be extended to test MDBS when it uses the secondary-memory-based directory management process.

LIST OF REFERENCES

1. Naval Postgraduate School Report NPS52-83-006, The Design and Analysis of a Multi-backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I), by Hsiao, David K., and Menon, Jaishankar, June, 1983.

2. Naval Postgraduate School Report NPS52-83-007, The Design and Analysis of a Multi-backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II), by Hsiao, David K., and Menon, Jaishankar, June, 1983.

3. Naval Postgraduate School Report NPS52-83-008, The Implementation of a Multi-backend Database System (MLBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS, by Kerr, Douglas S., Orooji, Ali, Shi, Zong-Zhi, and Strawser, Paula, June, 1983.

4. Naval Postgraduate School Report NPS52-82-008, The Implementation of a Multi-backend Database System (MLBS): Part II - The First Prototype MDBS and the Software Engineering Experience, by Higashida, Xingui Fe, Hsiao, David K., Kerr, Douglas S., Orooji, Ali, Shi, Zong-Zhi, and Strawser, Paula, June, 1982.

5. Naval Postgraduate School Report NPS52-83-003, The Implementation of a Multi-backend Database System (MLBS): Part III - The Message-Oriented Version with Concurrency Control and Secondary-Memory-Based Directory Management, by Boyne, Richard D., Demurjian, Steven A., Hsiao, David K., Kerr, Douglas S., and Orooji, Ali, March, 1983.

6. Naval Postgraduate School Report NPS52-84-005, The Implementation of a Multi-backend Database System (MLBS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-Process and Inter-Computer Messages, by Demurjian, Steven A., Hsiao, David K., Kerr, Douglas S. and Orooji, Ali, February, 1984.

7. Naval Postgraduate School Report NPS52-84-004, A Methodology for Benchmarking Relational Database Machines, by Strawser, Paula R., January, 1984.

8. University of Wisconsin Report MCS82-01870, Can Database Machines Do Better? A Comparative Performance Evaluation, by Bitton, Dina, DeWitt, David J., Turbyfill, Carolyn, December, 1983.

9. Kovalchik, Joseph G., Performance Evaluation Tools for a Multi-backend Database System, M.S. Thesis, Naval Postgraduate School, Monterey, California, December, 1983.

10. Database Machines, A Message-Oriented Implementation of a Multi-backend Database System (MDBS), by Boyne, Richard D., Hsiao, David K., Kerr, Douglas S., Orooji, Ali, September, 1983.

11. Concurrency Control in a Multi-backend Database System (MIDS), by Demurjian, Steven A., Hsiao, David K., Kerr, Douglas S., Menon, Jai, and Orooji, Ali, unpublished, March 1983.

12. An Attribute-Based System as a Database Kernel of Database Systems, by Demurjian, Steven A., Hsiao, David K., Macy, Griffen N., Strawser, Paula R., unpublished, March 1984.

13. Hsiao, David K. and Harary, F., "A Formal System for Information Retrieval From Files", Communications of The ACM, Vol. 13, No. 2, February 1970.

14. PCI11-B Parallel Communication Link Differential TDM Bus, Digital Equipment Corporation, Maynard, Mass., 1979.

BIBLIOGRAPHY

Hancock, Les and Krieger, Morris, The C Primer, McGraw-Hill Book Company, N.Y., 1983.

Kernnigan, Brian and Fitchie, Dennis M., The C Programming Language, Prentice-Hall, 1978.

RSX-11M/M-PLUS Executive Reference Manual, AA-H265A-TC, Digital Equipment Corporation, Maynard, Mass., 1979.

VAX/VMS System Services Reference Manual, AA-D018E-TE, Digital Equipment Corporation, Maynard, Mass., 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Dudley Knox Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	6
4.	Commandant of the Marine Corps Code CC Headquarters, Marine Corps Washington, D. C. 20380	1
5.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93943	1
6.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943	1
7.	Robert Tekampe 13913 Gum Lane Woodbridge, Virginia 22193	2
8.	Robert Watson 3481 Lyon Park Court Woodbridge, Virginia 22192	2

210813

Thesis
T237 Tekampe
c.1 Internal and external performance measurement methodologies for database systems.